Laboratorium 1

- **Podstawy systemu UNIX (1)**

1. Poruszanie się po drzewie katalogowym.

W systemie Unix wszystkie pliki są zorganizowane w pojedyncze drzewo katalogów. Korzeniem drzewa jest katalog o nazwie **/** . Każdy plik (katalog) w Unix'ie może być opisany ścieżką bezwzględną, która podaje lokalizację tego pliku (katalogu) w odniesieniu do korzenia drzewa, np:

```
/usr/local/include/gnu/java/beans/EmptyBeanInfo.h
/usr/bin/sh
/usr/local/include
```

Ścieżka bezwzględna zaczynia się zawsze od znaku **/** . Każdy proces (program) działający w systemie Unix posługuje się również pojęciem katalogu bieżącego. Ścieżka względna jest alternatywnym sposobem określenia lokalizacji pliku (względem katalogu bieżącego). Jej charakterystyczną cechą jest to, że NIE zaczyna się ona od znaku **/** . I tak będąc w katalogu **/usr/local/include** możemy odwołać się do wspomnianych wyżej przykładowych plików (katalogów) jako:

```
gnu/java/beans/EmptyBeanInfo.h
../../bin/sh
.
```

W drugim przypadku nazwa **..** (dwie kropki) oznacza katalog nadrzędny (o jeden poziom wyżej).
W trzecim przypadku nazwa **.** (kropka) oznacza „ten katalog".

Zmiana katalogu bieżącego jest możliwa za pomocą komendy **cd**

Przykłady użycia	
`cd KAT`	– przejście do katalogu KAT
`cd ..`	– przejście do katalogu nadrzędnego
`cd ~username`	– przejście do katalogu domowego dowolnego użytkownika
`cd`	– przejście do własnego katalogu domowego

Inne komendy:
- wyświetlenie bieżącej ścieżki: **pwd**
- utworzenie pustego pliku: **touch nazwa_pliku**
- skasowanie pliku: **rm nazwa_pliku**
- utworzenie katalogu: **mkdir nazwa_katalogu**
- skasowanie katalogu: **rmdir nazwa_katalogu**
- skasowanie katalogu z podkatalogami: **rm -rf nazwa_katalogu**
- skopiowanie pliku **a** do pliku **b** albo skopiowanie pliku **a** do katalogu **b**: **cp a b**
- skopiowanie katalogu **a** z podkatalogami do katalogu **b**: **cp -r a b**
- zmiana nazwy pliku **a** na **b** albo przeniesienie pliku/katalogu **a** do katalogu **b**: **mv a b**
- wyświetlenie zawartości pliku tekstowego: **less nazwa_pliku**

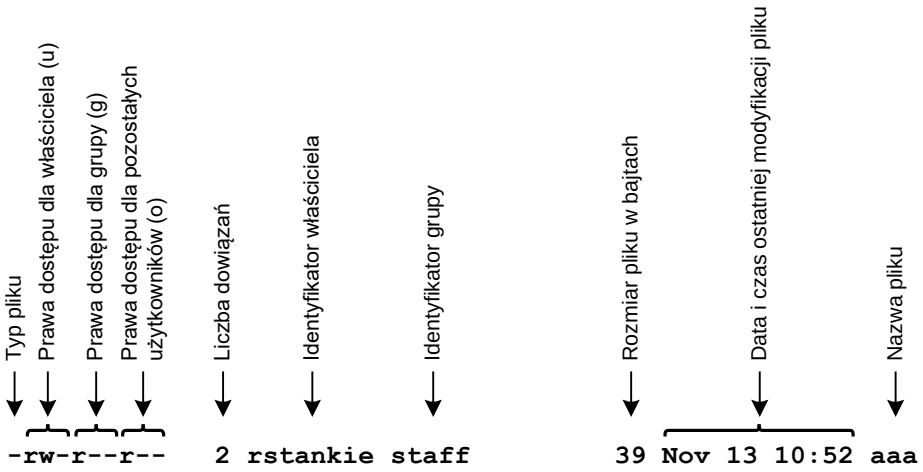
Wyświetlenie zawartości katalogu jest możliwe za pomocą polecenia **ls**

Wybrane opcje polecenia **ls**	
`-l`	– wyświetlenie informacji szczegółowych
`-a`	– wyświetlenie wszystkich plików (z ukrytymi włącznie, pliki ukryte w systemie Unix to pliki, których nazwy zaczynają się od kropki, np. `.bash_history`)
`-i`	– wyświetlenie numerów i-nodów (węzłów plików)
`-m`	– lista plików oddzielonych przecinkami
`-1`	– lista plików (tylko nazwy) w jednej kolumnie
`-d katalog`	– wyświetlenie informacji o katalogu. Bez opcji `-d` (`ls katalog`) zostałaby wyświetlona zawartość katalogu `katalog`, a nie informacje o nim samym.

Format wydruku dla `ls -l`

```
type            u     g     o    link owner group size(B) time name
{d,b,c,l,p,s,-}{rwxs}{rwxs}{rwxt}
```

- `type` – typ pliku:
  - `-` plik zwykły
  - `d` katalog
  - `b` specjalny plik blokowy
  - `c` specjalny plik znakowy
  - `l` link symboliczny
  - `p` pipe - kolejka FIFO
  - `s` socket
- prawa dostępu dla właściela (`u`), grupy (`g`) oraz pozostałych użytkowników (`o`)
  - `rwx` dopuszczone prawo odczytu/zapisu/wykonywania
  - `-` brak prawa
  - `s` modyfikacja efektywnego identyfikatora grupy lub użytkownika (patrz zajęcia 3)
  - `t` bit „przyklejenia” *sticky bit*
- `link` – liczba dowiązań (linków) twardych do pliku
- `owner` – identyfikator właściciela (zwykle nazwa użytkownika, w niektórych systemach identyfikator numeryczny)
- `group` – identyfikator grupy (zwykle nazwa grupy, w niektórych systemach identyfikator numeryczny)
- `size` - rozmiar pliku w bajtach
- `time` – data i czas ostatniej modyfikacji pliku
- `name` – nazwa pliku lub sepcyfikacja linku symbolicznego



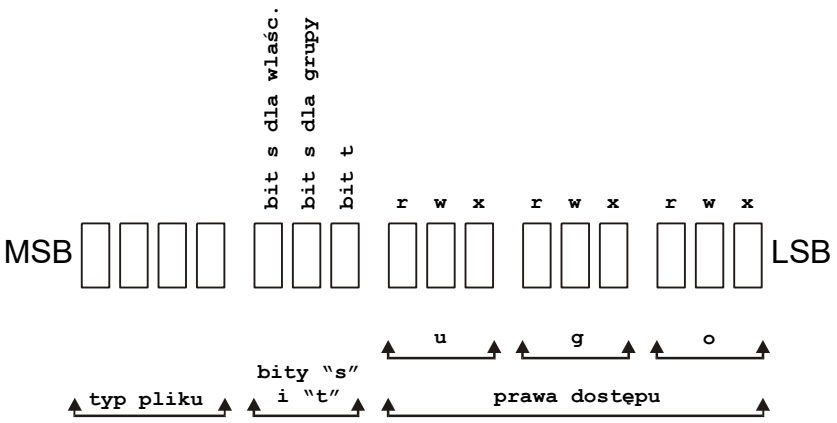
Przykład 1

Poniżej przedstawiono przykładowy wynik wykonania polecenia `ls -la`

```
> ls -la
total 64
drwxr-xr-x   3 rstankie staff        512 Nov 13 11:45 .
drwxr-xr-x   8 rstankie staff        512 Dec 18 16:04 ..
drwxr-xr-x   2 rstankie staff        512 Nov 13 11:31 KATALOG
-rw-r--r--   2 rstankie staff         39 Nov 13 10:52 aaa
-rw-r--r--   2 rstankie staff         39 Nov 13 10:52 bbb
lrwxrwxrwx   1 rstankie staff          3 Nov 13 10:51 ccc -> aaa
prw-r--r--   1 rstankie staff          0 Nov 13 10:50 pipe
-rw-r--r--   1 rstankie staff         22 Nov 13 10:50 plik zwykly
-rwxr-s---   1 rstankie staff      25496 Nov 13 10:50 type
```

2. Prawa dostępu do plików

1. Prawa dostępu do pliku są przechowywane w słowie trybu dostępu do pliku. Znacznie poszczególnych bitów przedstawiono poniżej.



Po trzy bity przeznaczono na określenie praw dostępu odpowiednio dla właściciela, grupy i pozostałych. Ustawienie danego bitu oznacza nadanie prawa dostępu, zaś skasowanie – odebranie.
Bity „s" w niektórych systemach można ustawiać niezależnie od tego czy nadano prawo uruchamiania. Często jednak ustawienie bitu „s" (dla użytkownika lub grupy) jest możliwe tylko wtedy gdy ustawiono prawo uruchamiania (odpowiednio dla właściciela lub grupy). W tych systemach odebranie prawa uruchamiania pociąga za sobą automatyczne skasowanie bitu „s".
Cztery bity przeznaczono na określenie typu pliku. Wartości odpowiadające poszczególnym typom plików będą omówione w dalszej części materiałów.

Przykład 2

a) Przykładowy opis pliku wyświetlony poleceniem `ls`.

```
> ls -l progr
-rwsr-sr-x   1 rstankie staff       1786 Dec 20 13:38 progr
```

Interpretacja jest następująca: ustawiono bit „s" dla właściciela grupy i nadano prawo wykonania „x"; ustawiono bit „s" dla grupy i nadano prawo wykonania grupie.

b) Poniżej przedstawiono przykładowy opis pliku wyświetlony poleceniem `ls` w systemie , w którym można ustawić bit „s" niezależnie od ustawienia prawa uruchamiania.

```
> ls -l progr
-rwSr-sr-x   1 rstankie staff       1786 Dec 20 13:38 progr
```

Interpretacja jest następująca: ustawiono bit „s" dla właściciela, ale nie nadano mu wcześniej prawa uruchamiania „x"; ustawiono bit „s" dla grupy i nadano prawo wykonania grupie.

2. Do zmieniania praw dostępu do plików służy polecenie `chmod`. Jego składnia jest następująca:

```
chmod {ugoa}{+-=}{rwxst},{} plik1 plik2 ...
```

Przykład 3

Poniżej przedstawiono różne sposoby użycia polecenia `chmod` wraz z komentarzem.

`chmod g-w ala`	odebranie grupie prawa zapisu do pliku `ala`
`chmod a+x,u+s ala`	nadanie prawa uruchamiania pliku `ala` właścicielowi, grupie i pozostałym oraz ustawienie bitu „s" dla właściciela
`chmod ug=rw a*c`	ustawienie właścicielowi i grupie praw `rw-` do wszystkich plików, których nazwa zaczyna się od litery `a` i kończy na `c`.

`chmod u=rw,go=r ala`
lub
`chmod 644 ala`
	ustawienie praw dostępu: `rw-r--r--` do pliku `ala`. Liczba 644 jest to zapis praw dostępu w postaci liczby w kodzie ósemkowym. Prawa `rw-r--r--` w zapisie bitowym to 110100100 co po zamianie na kod ósemkowy daje 644.

W ostatnim poleceniu chmod użyto numerycznej reprezentacji praw dostępu. Możliwe jest łatwe interpretowanie takiego zapisu jeżeli zapamiętamy, że trzy cyfry ósemkowe odpowiadają kolejno prawom `ugo` (user, group, others), a każda z cyfr powstaje przez dodanie 4(r) + 2(w) + 1(x). Przykładowo 751 to: 7=4(r)+2(w)+1(x); 5=4(r)+1(x) zatem prawa do pliku to rwxr-x--x.

Przykład 4

a) System, w którym do ustawienia bitu „s" konieczne jest nadanie prawa uruchamiania.

```
> ls -l progr
-rwxr--r-x   1 rstankie staff       1786 Dec 20 13:38 progr
> chmod g+s progr
chmod: WARNING: Execute permission required for set-ID on execution
for progr
> ls -l progr
-rwxr--r-x   1 rstankie staff       1786 Dec 20 13:38 progr
```

b) Systemie, w którym można ustawić bit „s" niezależnie od ustawienia prawa uruchamiania.

```
> ls -l progr
-rwxr--r-x   1 rstankie staff       1786 Dec 20 13:38 progr
> chmod g+s progr
> ls -l progr
-rwxr-Sr-x   1 rstankie staff       1786 Dec 20 13:38 progr
```

3. Domyślne prawa dostępu i maska praw dostępu

Każdemu tworzonemu plikowi na dysku nadawane są domyślne prawa dostępu. Zależą one od bieżącego ustawienia maski praw dostępu. Do wyświetlenia aktualnej maski praw dostępu (oraz zmieniania maski) służy polecenie **`umask`**. Ustawienie jakiegoś bitu w masce na 1 oznacza, że dane prawo dostępu ma być domyślnie zabronione.

```
> umask
22
```

Liczba 22 oznacza, że aktualna maska jest 022 czyli domyślnie zabronione jest prawo zapisu dla grupy i pozostałych. Przy tak ustawionej masce nowo utworzony plik będący programem wykonywalnym (np. wynik kompilacji) otrzyma prawa dostępu **`rwxr-xr-x`**. Natomiast nowo utworzony plik nie będący programem wykonywalnym (np. utworzony poleceniem **`touch`**) otrzyma prawa dostępu **`rw-r--r--`**.

Zmiana maski nie powoduje zmiany praw dostępu plików już istniejących.

3. System plików

W systemie UNIX każdy zbiór informacji lub urządzenie jest traktowane jako plik. Rozróżnia się następujące typy plików:

- plik zwykły
- katalog
- specjalny plik blokowy
- specjalny plik znakowy
- link symboliczny
- pipe - kolejka FIFO
- gniazdo

Każdy plik jest reprezentowany przez strukturę informacyjną nazywaną węzełem pliku (i-node). Każdy plik jest jednoznacznie identyfikowany przez węzeł. W jednym systemie plików nie ma dwóch plików o tym samym numerze węzła. Zasadniczo przez „plik" będziemy więc rozumieć konkretny węzeł. Przez utworzenie nowego pliku rozumiemy utworzenie nowego węzła.

W węźle pliku jest zapisany typ pliku oraz parametry dostępu użytkownika, grupy i pozostałych osób. Nie istnieje atrybut informujący o tym czy plik jest ukryty, czy nie (w systemie UNIX nie istnieje odpowiednik bajtu atrybutów pliku z systemu DOS/Windows). Ukrycie pliku jest możliwe przez nadanie mu nazwy rozpoczynającej się od znaku kropki '`.`'. To czy plik jest wykonywalny nie zależy ani od rozszerzenia, ani od zawartości pliku, a jedynie od praw dostępu - ten kto próbuje uruchomić (wykonać) plik musi mieć takie uprawnienia.

Nazwy plików w systemie UNIX mogą zawierać zarówno duże jak i małe litery (są one rozróżniane). Znak kropki może wystąpić w nazwie i to wielokrotnie. Zwyczajowo przyjęto pewne nadawania rozszerzeń pewnym rodzajom plików np. :

- `.c` plik zawierający kod źródłowy programu w języku C
- `.C` albo `.cc` plik zawierający kod źródłowy programu w języku C++
- `.out` plik zawierający program wynikowy kompilacji
- `.asm` plik zawierający kod źródłowy programu w asemblerze danego systemu

W systemie UNIX istnieje możliwość tworzenia wielu dowiązań do jednego pliku. Dowiązanie jest utworzeniem nowej nazwy wskazującej na ten sam plik (węzeł). Są to tzw. linki twarde (dowiązania twarde). W wyniku utworzenia dowiązania twardego nie jest tworzony nowy węzeł na dysku. Następuje jedynie dodanie wpisu w jakimś katalogu (powiązanie nazwa–numer węzła). Aktualna liczba dowiązań jest przechowywana w węźle. Usunięcie węzła z systemu następuje w momencie skasowania wszystkich istniejących dowiązań twardych, o ile dany plik nie jest otwarty przez jakiś proces (usunięcie węzła nastąpi wówczas w momencie zamknięcia pliku).

Można też tworzyć dowiązania symboliczne (zwane również miękkimi). Dowiązanie symboliczne jest plikiem specjalnym (osobnym węzłem!) zawierającym ścieżkę dostępu do innego pliku. Utworzenie dowiązania symbolicznego wiąże się więc z utworzeniem nowego pliku (węzła).

Każdy katalog (plik-katalog) ma co najmniej dwa dowiązania (twarde). Pierwszym jest jego nazwa własna, a drugim nazwa „`.`" wewnątrz tego katalogu. Jeżeli katalog ma podkatalogi to w każdym z podkatalogów jest plik „`..`" wskazujący na katalog macierzysty. Liczba dowiązań do katalogu będzie więc równa liczbie jego podkatalogów + 2.

4. Zmienne środowiskowe

Każdy użytkownik systemu ma ustawiony zestaw zmiennych środowiskowych. Przechowują one między innymi informacje o serwerze, systemie, użytkowniku, typie terminala i ścieżkach. Istnieje możliwość definiowania własnych zmiennych środowiskowych. Listę zmiennych środowiskowych można wyświetlić wykonując polecenie `env` lub `set`. Do zmieniania wartości zmiennych środowiskowych oraz ustawiania nowych zmiennych służy polecenie `setenv` dla shella z rodziny csh (csh, tcsh):

```
setenv ZMIENNA wartość
```

W przypadku shella z rodziny sh (sh, bash) zmienne środowiskowe ustawia się pisząc

```
export zmienna=wartość
```

Przykład 5

Poniżej przedstawiono przykładową listę zmiennych środowiskowych. Ustawiono też nową zmienną o nazwie ZMIENNA i nadano jej wartość TEST.

```
> setenv ZMIENNA test
> env
HOME=/export/home/rstankie
INCLUDE=/usr/include:/usr/local/include
LD LIBRARY PATH=/usr/lib/sparcv9:/usr/lib:/etc/lib:/usr/local/pgsql/lib:
LOGNAME=rstankie
&1MAIL=/var/mail/rstankie
MANPATH=/usr/dt/man:/usr/openwin/man:/opt/SUNWspro/man:/usr/share/man:/usr/local/pgsql/man
PATH=/opt/local/bin:/sbin:/usr/ucc/bin:/usr/ccs/bin:/usr/sbin:/opt/sbin:/export/bin:/bin:/usr/
local/pgsql/bin:/bin:/usr/bin:/usr/local/bin
SHELL=/bin/sh
SSH CLIENT=149.156.98.60 3574 22
SSH TTY=/dev/pts/3
TERM=vt100
TZ=Poland
USER=rstankie
HOSTTYPE=sun4
VENDOR=sun
OSTYPE=solaris
MACHTYPE=sparc
SHLVL=1
PWD=/export/home/rstankie
GROUP=staff
HOST=pluton.kt.agh.edu.pl
REMOTEHOST=student.uci.agh.edu.pl
ZMIENNA=test
```

5. Korzystanie z pomocy (man)

Większość poleceń systemowych, poleceń shella oraz funkcji języka C jest opisanych w tekstach pomocy (manualach). Aby wyświetlić tekst pomocy należy wykonać polecenie `man temat`. Przykładowo, chcąc szukać informacji na temat składni polecenia `ls` wykonujemy polecenie `man ls`. Manuale można też przeszukiwać po słowach kluczowych. Wykonuje się wówczas polecenie `man -k slowo_kluczowe`.

Z korzystaniem z polecenia man oraz innych form dokumentacji związany jest często używany akronim „RTFM”. Sugerujemy samodzielne sprawdzenie znaczenia tego skrótu korzystając np. z http://www.wikipedia.org.

6. Pytania kontrolne

1. Jak sprawdzić jaki jest aktualnie katalog bieżący? Jak go zmienić?
2. Jak wyświetlić listę plików?
3. Jak działają opcje -l oraz -a w poleceniu ls?
4. Zinterpretuj następujący wynik działania polecenia `ls -l`

```
-rw-r--r--   2 rstankie staff         39 Nov 13 10:52 bbb
```

5. Plik `.` (kropka) - co to jest?
6. Plik `..` (dwie kropki) - co to jest?
7. Co robi polecenie `chmod 750 test`
8. Co robi polecenie `umask 002`
9. W jaki sposób można wyświetlić aktualne zmienne środowiskowe?
10. Jakim poleceniem można:
    - skopiować plik?
    - zmienić nazwę pliku?
    - usunąć plik/katalog/katalog z podkatalogami?
    - utworzyć pusty plik
    - wyświetlić zawartość pliku tekstowego?

11. Jak jest różnica pomiędzy poleceniami `set` i `env` ?

7. Ćwiczenia

Uruchom maszynę wirtualną AKISO/SO, zaloguj się jako student i otwórz okno terminala.

Ćwiczenie 1:
Wykonaj poniższe operacje. Za każdym razem użyj jednego polecenia Unix'a.

- zmień katalog bieżący na **`/usr/local/sbin`** używając ścieżki **bezwzględnej**
  - sprawdź, jaki jest katalog bieżący
- zmień katalog bieżący na **`/usr/sbin`** używając ścieżki **bezwzględnej**
  - sprawdź, jaki jest katalog bieżący
- zmień katalog bieżący na **`/usr/local/sbin`** używając ścieżki **względnej**
  - sprawdź, jaki jest katalog bieżący
- zmień katalog bieżący na **`/usr/sbin`** używając ścieżki **względnej**
  - sprawdź, jaki jest katalog bieżący

Ćwiczenie 2:
Upewnij się, że jesteś w katalogu **`/usr/sbin`**
Wykonaj poniższe polecenia obserwując jakie otrzymujesz wyniki.

- wykonaj polecenie **`ls`**
- wykonaj polecenie **`ls -a`**
- wykonaj polecenie **`ls -m`**
- wykonaj polecenie **`ls -1`**
- wykonaj polecenie **`ls -d`**
- wykonaj polecenie **`ls -l`**
- wykonaj polecenie **`ls -la`**

Polecenie **`ls -l`** generuje więcej tekstu niż się mieści na jednym ekranie. Żeby obejrzeć cały wydruk, przekieruj *standardowe wyjście* polecenia do polecenia **`less`**, które wyświetla tekst po jednej stronie na raz.
Wykonaj to komendą **`ls -l | less`**
Spróbuj wykonać inne z powyższych poleceń w ten sposób.

W poleceniu **`less`** możesz użyć klawiszy:

- `q` - kończy działanie polecenia **`less`**
- `Enter` - przewija o 1 linię w przód
- `Spacja` - przewija o 1 ekran w przód
- `y` - przewija o 1 linię w tył
- `b` - przewija o 1 ekran w tył
- `/` - szuka tekstu w przód (np. wpisz `/rm` i naciśnij Enter, wpisanie `/` i naciśnięcie Enter szuka dalej tego samego tekstu)
- `?` - szuka tekstu wstecz (np. wpisz `?ls` i naciśnij Enter, wpisanie `?` i naciśnięcie Enter szuka dalej tego samego tekstu w tył)

Ćwiczenie 3:

- wyświetl dokumentację polecenia ls
- do przeglądania tekstu użyj skrótów klawiaturowych polecenia less
- przewiń tekst o 1 stronę do przodu/do tyłu
- przewiń tekst o 1 linię do przodu/do tyłu
- wyszukaj ciąg znaków dir
- poszukaj tego samego ciągu kilka razy w przód, następnie kilka razy w tył

Ćwiczenie 4:
Wykonaj polecenie **`cd`**. W jakim katalogu jesteś? Jakie są pliki w bieżącym katalogu?

- wykonaj polecenie **`ls -l /usr/sbin`**
- wykonaj polecenie **`ls -ld /usr/sbin`**

Ćwiczenie 5:

- Upewnij się, że jesteś w katalogu domowym
- Utwórz pusty plik o nazwie `plik1`
- Sprawdź, czy plik się utworzył. Kto jest właścicielem pliku? Jakie są jego prawa dostępu? Jaki jest rozmiar pliku? Ile jest dowiązań do tego pliku?
- Skasuj utworzony plik.
- Utwórz katalog o nazwie `kat1`
- Sprawdź, czy katalog się utworzył. Kto jest jego właścicielem, jakie są jego prawa dostępu? Jaki jest rozmiar katalogu? Ile jest dowiązań do tego katalogu?
- W katalogu kat1 utwórz plik o nazwie `plik2` i katalog `kat2`
- Usuń katalog `kat2` za pomocą polecenia **`rmdir`**.
- Spróbuj usunąć katalog kat1 za pomocą polecenie **`rmdir`**. Dlaczego się nie udało?
- Usuń katalog kat1 za pomocą polecenia **`rm -rf`**

Ćwiczenie 6:

- Obejrz zawartość pliku `/etc/passwd`
- Skopiuj ten plik do katalogu domowego. Sprawdź, czy plik się skopiował.
- Zmień nazwę skopiowanego pliku na test
- Utwórz katalog kat3
- Przenieś skopiowany plik do tego katalogu
- Skopiuj katalog `/etc/rc.d/init.d` do katalogu domowego. Sprawdź, czy katalog się skopiował
- Zmień nazwę skopiowanego katalogu z init.d na test.d
- Przenieś katalog test.d do katalogu kat3

Pisząc polecenia w UNIX'ie warto znać użyteczne kombinacje klawiszy:

- CTRL-c przerywa wykonywanie bieżącego polecenia
- CTRL-d kończy wprowadzanie tekstu z klawiatury
- klawisz TAB „rozwija" nazwy plików, dwukrotne naciśnięcie TAB wyświetla możliwe rozwinięcia
- lewy/prawy klawisz strzałki służą do edycji polecenia
- CTRL-a oraz CTRL-e skaczą na początek (koniec) bieżącej linii
- klawisze strzałek góra/dół przywołują poprzednio wydane polecenia
- CTRL-r szuka w poprzednio wydanych poleceniach zadanego tekstu
- CTRL-s blokuje wyświetlanie tekstu na ekranie, CTRL-q odblokowuje - tę kombinację warto znać głównie ze względu na możliwość przypadkowego naciśnięcia CTRL-s w trakcie pracy (CTRL-s/CTRL-q nie działają w oknie X-Windows, zamiast nich należy użyć klawisza scroll lock)

Ponadto warto pamiętać, że zaznaczenie myszką jakiegoś tekstu i naciśnięcie środkowego klawisza myszy wkleja ten tekst do okna.

Ćwiczenie 7:

- przejdź do konsoli tekstowej linux'a naciskając CTRL-ALT-F2 i zaloguj się jako student
- polecenie **`ls -lR /`** wyświetla wszystkie pliki w danym systemie unix - jest tego bardzo dużo - spróbuj kombinacji klawiszy CTRL-s CTRL-q CTRL-c
- wróć do konsoli graficznej naciskając CTRL-ALT-F7 (dla SO CTRL-ALT-F1)
- w oknie w którym wydawałeś wcześniej wszystkie polecenia wciśnij **`CTRL-r`** następnie wpisz **`mk`** a następnie naciśnij strzałkę w lewo. Zobacz jakie polecenie wyszukałeś
- wykonaj komendę **`ls -l /usr/share/X11/locale/iso8859-2/Compose`** wpisując na początek **`ls -l /u`** i używając klawisza TAB do uzupełnienia ścieżki. Dopisuj tylko niezbędne znaki (dwukrotne naciśnięcie TAB to lista możliwości).
- przywołaj poprzednie polecenie i przetestuj kombinacje CTRL-a CTRL-e
- umieść kursor w środku linii i naciśnij Enter - zapamiętaj, że NIE MUSISZ ustawiać kursora na końcu linii przed wykonaniem polecenia

Ćwiczenie 8:

- utwórz grupę staff za pomocą komendy **`groupadd staff`**
- utwórz użytkownika test1 za pomocą komendy **`useradd -m test1 -g users`**
- utwórz użytkownika test2 za pomocą komendy **`useradd -m test2 -g users`**
- utwórz użytkownika test3 za pomocą komendy **`useradd -m test3 -g staff`**
- Użytkownicy test1 i test2 są w grupie users natomiast użytkownik test3 jest w grupie staff
- ustaw jakieś hasła nowym użytkownikom (może być to samo)
    - **`passwd test1`**
    - **`passwd test2`**
    - **`passwd test3`**
- otwórz trzy konsole (okna terminala), za pomocą komendy **`su - test1`** zaloguj w pierwszej konsoli użytkownika test1, podobnie zaloguj użytkowników test2 i test3 w drugim i trzecim oknie
- sprawdź jakie są katalogi domowe użytkowników test1, test2 i test3
- wyświetl zawartość i prawa dostępu katalogów tych użytkowników - zauważ różnicę w przypisaniu do grup
- ustaw prawa dostępu 755 na katalogach domowych użytkowników test1, test2 i test3

Prawa dostępu

Prawa dostępu składają się z praw właściciela pliku, praw grupy (prawa grupy nigdy nie dotyczą właściciela pliku, nawet jeżeli właściciel jest jednocześnie członkiem tej grupy) oraz praw pozostałych użytkowników (nie będących ani właścicielem, ani członkami grupy do której jest przypisany plik).

Dla plików:

r - prawo odczytu zawartości pliku
w - prawo zapisu/zmiany zawartości pliku
x - prawo wykonania pliku
bity s i t zostaną omówione na jednych z kolejnych zajęć

Dla katalogów:

r - prawo odczytu zawartości katalogu (spisu plików)
w - prawo tworzenia nowych plików i kasowania istniejących oraz zmian nazw plików
x - prawo wejścia do katalogu
s dla grupy - nowo tworzone pliki i katalogi odziedziczą grupę z tego katalogu (niezależnie od tego, kto je utworzy)
t dla pozostałych - kasowanie i zmiana nazwy plików (i katalogów) w katalogu na którym ustawiono bit t jest możliwa tylko przez właściciela katalogu, właściciela pliku oraz root'a. W niektórych systemach operacyjnych (Solaris) kasowanie i zmiana nazwy danego pliku jest również możliwe przez wszystkich użytkowników, którzy mają prawo zapisu tego pliku.

Ćwiczenie 9:

- jako użytownik test1 utwórz w swoim katalogu domowym plik o nazwie test1
- ustaw prawa dostępu na tym pliku tak, aby użytkownik test1 mógł czytać i pisać do tego pliku, grupa i inni nie mogli ani czytać, ani pisać do tego pliku - użyj zapisu literowego praw dostępu
    - przetestuj zapis do pliku test1 za pomocą komendy pwd > test1 (> przekierowuje standardowe wyjście komendy pwd do wskazanego pliku)
    - przetestuj odczyt z pliku test1 (wyświetl go)
    - powtórz powyższe testy jako użytkownik test2 i test3
- jako użytkownik test1 dodaj na pliku test1 prawo zapisu dla grupy i odczytu dla pozostałych
- spróbuj czy możesz zmieniać prawa dostępu pliku test1 nie będąc jego właścicielem (jako użytkownik test2)
- przetestuj odczyt i zapis pliku test1 z użytkowników test1 test2 test3
- powtórz ćwiczenie używając numerycznej postaci praw dostępu

Ćwiczenie 10:

- jako użytownik test1
  - utwórz w swoim katalogu domowym katalog o nazwie kat1
  - ustaw prawa dostępu na tym katalogu tak, aby użytkownik test1 mógł wykonywać wszystkie operacje, a grupa i pozostali nie mogli wykonywać żadnych operacji
  - utwórz plik test2 w katalogu kat1, ustaw prawa 666 na tym pliku
  - zapisz coś do tego pliku i odczytaj to,
- zezwól na zapis dla grupy i odczyt dla pozostałych na katalogu kat1
  - jako użytkownik test2 (test3) - czy możesz wyświetlić zawartość katalogu kat1
  - dołóż prawo „x" dla grupy i pozostałych na katalogu kat1 i spróbuj ponownie wyświetlić zawartość katalogu kat1
  - czy użytkownicy test2 i test3 mogą czytać i pisać z/do pliku test2 ?
  - czy użytkownik test3 może zmienić nazwę pliku test2 albo skasować ten plik?
  - czy użytkownik test2 może zmienić nazwę pliku test2 albo skasować ten plik?

Ćwiczenie 11:

- jako użytownik test1 utwórz plik /tmp/test1
- jako użytownik test2 utwórz plik /tmp/test2 i spróbuj skasować plik /tmp/test1
- korzystając z konsoli root'a usuń bit t na katalogu /tmp
- jako użytownik test2 spróbuj ponownie skasować plik /tmp/test1
- korzystając z konsoli root'a ustaw ponownie bit t na katalogu /tmp

Ćwiczenie 12:

- jako użytownik test1 utwórz w swoim katalogu domowym katalog o nazwie kat
- ustaw prawa 777 na katalogu kat
- w katalogu kat
  - jako użytkownik test2 utwórz plik o nazwie test2 i katalog o nazwie kat2
  - jako użytkownik test3 utwórz plik o nazwie test3 i katalog o nazwie kat3
- kto jest właścicielem i jaka jest grupa obiektów utworzonych w katalogu kat
- ustaw prawo s dla grupy na katalogu kat
- w katalogu kat
  - jako użytkownik test2 utwórz plik o nazwie test2s i katalog o nazwie kat2s
  - jako użytkownik test3 utwórz plik o nazwie test3s i katalog o nazwie kat3s
- kto jest właścicielem i jaka jest grupa nowo utworzonych obiektów utworzonych w katalogu kat
- jakie są prawa katalogów utworzonych w katalogu kat

Ćwiczenie 13:

- użyj polecenia umask tak, aby nowo tworzone katalogi miały prawa rwxr-x--x
- przetestuj

Ćwiczenie 14:
Poniżej przedstawiono przykładową sesję na serwerze. Polecenia i ich rezultaty opatrzono komentarzami. Wykonaj te same polecenia.

```
> cd
> mkdir KAT3
> cd KAT3
> w > a
```

Utworzenie pliku `a` przez przekierowanie wyników wykonania polecenia `w`.

```
> ls -lia
total 3
    448009 drwxr-xr-x   2 rstankie staff      512 Dec 20 14:32 .
    342609 drwxr-xr-x  11 rstankie staff      512 Dec 20 14:29 ..
    448010 -rw-r--r--   1 rstankie staff      228 Dec 20 14:32 a
```

Plik o nazwie `a` ma numer węzła 448010. Plik „`.`" jest to katalog bieżący. Czas modyfikacji pliku „`.`" jest taki sam jak czas modyfikacji pliku `a`, gdyż w momencie utworzenia pliku `a` został również utworzony wpis w pliku „`.`" zawierający powiązanie nazwy pliku `a` z jego węzłem.

```
> cp a b
> ls -lia
total 4
    448009 drwxr-xr-x   2 rstankie staff      512 Dec 20 14:34 .
    342609 drwxr-xr-x  11 rstankie staff      512 Dec 20 14:29 ..
    448010 -rw-r--r--   1 rstankie staff      228 Dec 20 14:32 a
    448011 -rw-r--r--   1 rstankie staff      228 Dec 20 14:34 b
```

Skopiowano plik `a` do pliku `b`. Plik `b` jest nowym plikiem o numerze węzła 448011 i jest wierną kopią pliku `a`. Ma inny (przydzielony przez system w dowolny sposób) numer węzła. Czas modyfikacji pliku „`.`” j.w.

```
> mv a c
> ls -lia
total 4
    448009 drwxr-xr-x   2 rstankie staff      512 Dec 20 14:38 .
    342609 drwxr-xr-x  11 rstankie staff      512 Dec 20 14:29 ..
    448011 -rw-r--r--   1 rstankie staff      228 Dec 20 14:34 b
    448010 -rw-r--r--   1 rstankie staff      228 Dec 20 14:32 c
```

Poleceniem `mv` zmieniono nazwę pliku `a` na `c`. Czas modyfikacji tego pliku oraz numer węzła (448010) się nie zmienił. Zmiana nazwy pliku to jedynie zmiana wpisu w katalogu (czas modyfikacji pliku „`.`” zmienił się).

```
> ln c a
> ls -lia
total 5
    448009 drwxr-xr-x   2 rstankie staff      512 Dec 20 14:41 .
    342609 drwxr-xr-x  11 rstankie staff      512 Dec 20 14:29 ..
    448010 -rw-r--r--   2 rstankie staff      228 Dec 20 14:32 a
    448011 -rw-r--r--   1 rstankie staff      228 Dec 20 14:34 b
    448010 -rw-r--r--   2 rstankie staff      228 Dec 20 14:32 c
```

Poleceniem `ln` utworzono dowiązanie twarde o nazwie `a` do węzła pliku `c`. Należy zwrócić uwagę, że pliki `a` i `c` mają ten sam numer węzła. Jest to więc jeden plik. Nie zmienił się też czas modyfikacji tego pliku. Liczba dowiązań natomiast zmieniła się na 2. Ponadto zmienił się czas modyfikacji pliku „`.`”, gdyż utworzenie dowiązania twardego jest niczym innym jak utworzeniem nowego wpisu w pliku „`.`” (zawierającego powiązanie nowej nazwy z istniejącym węzłem).

```
> ln -s a d
> ls -lia
total 6
    448009 drwxr-xr-x   2 rstankie staff      512 Dec 20 14:46 .
    342609 drwxr-xr-x  11 rstankie staff      512 Dec 20 14:29 ..
    448010 -rw-r--r--   2 rstankie staff      228 Dec 20 14:32 a
    448011 -rw-r--r--   1 rstankie staff      228 Dec 20 14:34 b
    448010 -rw-r--r--   2 rstankie staff      228 Dec 20 14:32 c
    448012 lrwxrwxrwx   1 rstankie staff        1 Dec 20 14:46 d -> a
```

Poleceniem `ln -s` utworzono dowiązanie symboliczne o nazwie `d` do pliku `a`. Plik `d` jest nowym plikiem (numer węzła 448012). Plik ten zawiera ścieżkę dostępu do pliku `a`. Dlatego też jego rozmiar jest równy 1 bajt. Nie zmieniła się liczba dowiązań do pliku `a`.

```
> ln xx yy
ln: cannot access xx
```

Próbowano utworzyć dowiązanie twarde `yy` do nieistniejącego pliku `xx`. Wystąpił błąd, gdyż nie można tworzyć dowiązania twardego do nieistniejącego węzła.

```
> ln -s xx yy
> ls -lia
total 7
    448009 drwxr-xr-x   2 rstankie staff      512 Dec 21 10:27 .
    342609 drwxr-xr-x  11 rstankie staff      512 Dec 20 14:29 ..
    448010 -rw-r--r--   2 rstankie staff      228 Dec 20 14:32 a
    448011 -rw-r--r--   1 rstankie staff      228 Dec 20 14:34 b
    448010 -rw-r--r--   2 rstankie staff      228 Dec 20 14:32 c
    448012 lrwxrwxrwx   1 rstankie staff        1 Dec 20 14:46 d -> a
    448013 lrwxrwxrwx   1 rstankie staff        2 Dec 20 14:49 yy -> xx
> less yy
yy: No such file or directory
```

Próba utworzenia dowiązania symbolicznego do nieistniejącego pliku się powiodła. Utworzenie dowiązania symbolicznego jest tylko wykreowaniem pliku zawierającego ścieżkę dostępu. Ścieżka może wskazywać na nieistniejący plik. Próba odczytania pliku `yy` (polecenie `less`) spowoduje błąd, gdyż plik `xx`, na który wskazuje dowiązanie nie istnieje.

Ćwiczenie 15:

Wykonując następujący ciąg poleceń należy zwrócić uwagę na numery węzłów oraz liczby dowiązań. Kolorem czerwonym zaznaczono wszystkie dowiązania do pliku katalogu `KAT4` o numerze węzła 193611.

```
> cd
> mkdir KAT4
> ls -lia
    342609 drwxr-xr-x   6 rstankie staff        512 Dec 21 10:47 .
         2 drwxr-xr-x  34 root     root        1024 Dec 18 11:42 ..
     57931 drwxr-xr-x   2 rstankie staff        512 Dec 20 13:41 KAT1
    349641 drwxr-xr-x   2 rstankie staff        512 Dec 20 12:28 KAT2
    448009 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:27 KAT3
    193611 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:49 KAT4
> cd KAT4
> ls -lia
total 2
    193611 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:47 .
    342609 drwxr-xr-x   6 rstankie staff        512 Dec 21 10:47 ..
```

Polecenie `mkdir KAT4` spowodowało utworzenie nowego pliku specjalnego (katalogu) o numerze węzła 193611. Plik ten ma początkowo dwa dowiązania: nazwę własną `KAT4` oraz nazwę „`.`” wewnątrz katalogu `KAT4` (samego siebie).

```
> mkdir KT1
> ls -lia
total 3
    193611 drwxr-xr-x   3 rstankie staff        512 Dec 21 10:49 .
    342609 drwxr-xr-x   6 rstankie staff        512 Dec 21 10:47 ..
    285195 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:49 KT1
> cd KT1
> ls -lia
total 2
    285195 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:49 .
    193611 drwxr-xr-x   3 rstankie staff        512 Dec 21 10:49 ..
```

Wewnątrz katalogu `KAT4` utworzono katalog `KT1`. Liczba dowiązań do pliku 193611 wzrosła teraz do 3. Nowe dowiązanie do tego pliku to nazwa „`..`” wewnątrz katalogu `KT1`.

```
> cd ..
> mkdir KT2
> ls -lia
total 4
    193611 drwxr-xr-x   4 rstankie staff        512 Dec 21 10:49 .
    342609 drwxr-xr-x   6 rstankie staff        512 Dec 21 10:47 ..
    285195 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:49 KT1
    437832 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:49 KT2
> cd KT2
> ls -lia
total 2
    437832 drwxr-xr-x   2 rstankie staff        512 Dec 21 10:49 .
    193611 drwxr-xr-x   4 rstankie staff        512 Dec 21 10:49 ..
```

Wewnątrz katalogu `KAT4` utworzono kolejny katalog: `KT2`. Liczba dowiązań do pliku 193611 wzrosła teraz do 4. Nowe dowiązanie do tego pliku to nazwa „`..`” wewnątrz katalogu `KT2`.
Na poniższym rysunku przedstawiono powstałe drzewo katalogowe. Każda ramka to zawartość jednego katalogu. Kolorem czerwonym oznaczono wszystkie dowiązania wskazujące na ten sam plik o numerze węzła 193611 (katalog `KAT4`).

Ćwiczenie 16:

- wyświetl zmienne środowiskowe za pomocą polecenia, zwróć uwagę na zmienną PATH
- wykonaj PATH=/bin
- wykonaj polecenie ls
- wykonaj polecenie man
- wykonaj polecenie /usr/bin/man ls
- wykonaj PATH=/bin:/usr/bin
- wykonaj polecenie man ls

Zmienna PATH przechowuje informację o katalogach, w których znajdują się komendy.

Laboratorium 2

- **Podstawy systemu UNIX (2)**

1. Edycja tekstów

Podstawowym edytorem, który jest dostępny w większości odmian Unix'a jest edytor vi. Warto znać podstawy jego obsługi ze względu na to, że w danym systemie może być on jedynym dostępnym edytorem. Dodatkowo vi nie dodaje nic do plików „od siebie", a w szczególności nie zamienia tabulacji na spacje i nie dzieli wprowadzonych linii (nawet bardzo długich - rzędu kilku tysięcy znaków) co jest istotne w wielu plikach konfiguracyjnych. Rozszerzeniem vi jest vim, w którym nieco ułatwiono obsługę i dołożono interfejs graficzny (gvim). Oprócz vi bywają dostępne inne edytory, zarówno pracujące w trybie tekstowym (joe, pico) jak też graficznym (kate).

Większość edytorów można uruchomić zarówno bez podania nazwy pliku jak też podając od razu w linii komend nazwę pliku do edycji, np. **`vi /etc/passwd`**

Specyficzną cechą edytora vi są tryby pracy - tryb komend i tryb wprowadzania tekstu. Pierwszy tryb służy wyłącznie do poruszania się po tekście i wykonywania operacji na istniejącym tekście (wycinanie, wklejanie, kasowanie, szukanie tekstu), drugi tryb służy wyłącznie (!) do wpisywania nowego tekstu - poruszanie się po tekście jest niemożliwe. To ostatnie ograniczenie zostało usunięte w rozszerzonej wersji vi o nazwie vim, standardowo dostępnej w linuxie.

Zakończenie pracy z edytorem:

`:x`	wyjście z edytora z zapisem wprowadzonych poprawek
`:q!`	wyjście z edytora bez zapisywania wprowadzonych poprawek

Tryb wprowadzania tekstu:

`i`	umożliwia wpisanie tekstu przed pozycją kursora (insert)
`a`	umożliwia wpisanie tekstu za pozycją kursora (append)
`o`	(mało o) umożliwia pisanie tekstu w nowej linii poniżej bieżącej
`O`	(duże O) umożliwia pisanie tekstu w nowej linii powyżej bieżącej
`Esc`	(albo CTRL-c) kończy wprowadzanie tekstu zainicjowane komendami i, a, o, O oraz powoduje przejście do trybu wprowadzania komend

Modyfikowanie tekstu:

`r x`	zmienia znak na pozycji kursora na znak x (replace)
`R text Esc`	zmienia znaki od kursora począwszy na wpisywany text (nadpisuje)

Usuwanie i edycja tekstu:

`x`	usuwa znak wskazany przez kursor
`dd`	wycina bieżącą linię
`dNd`	wycina N linii poczynając od linii bieżącej
`yy`	kopiuje bieżącą linię
`yNy`	kopiuje N linii począwszy od bieżącej
`p`	wkleja skopiowane (za pomocą yy) lub wycięte (za pomocą dd) linie

Przesuwanie kursora i przeglądanie pliku:

`0`	przesuwa kursor do pierwszego znaku w linii
`$`	przesuwa kursor do ostatniego znaku w linii
`nG`	ustawia kursor na początku linii o numerze n
`G`	ustawia kursor na końcu zbioru
`Ctrl+F`	przesuwa tekst o 1 stronę do przodu
`Ctrl+B`	przesuwa tekst o 1 stronę do tyłu

Przeszukiwanie od bieżącej linii:

`/abcd`	przeszukuje plik i ustawia kursor na początku zadanego łańcucha znaków abcd

Inne komendy:

`:set nu`	przed każdą linią wyświetla jej numer
`:syntax on`	włącza kolorowanie składni
`:x,yw name`	zapisuje linie o numerach od x do y do pliku name
`:r name`	wczytuje plik name za bieżącą linię
`:w`	zapis pliku
`.`	(kropka) powtarza ostatnią komendę
`CTRL+v`	włącza tryp operacji blokowych

2. Strumienie, potoki (łącza komunikacyjne)

Każdy program uruchomiony w Unix'ie ma dostęp do następujących strumieni:

- stdin - standardowe wejście, domyślnie klawiatura
- stdout - standardowe wyjście - domyślnie ekran
- stderr - standardowe wyjście błądów - domyślnie ekran

Warto również wiedzieć, że wielu programach, w których należy podać nazwę pliku z danymi, użycie nazwy pliku – (minus) spowoduje użycie stdin albo stdout.

W momencie uruchamiania programu możliwa jest podmiana domyślnych dowiązań strumieni stdin/stdout/stderr na inne, za pomocą przekierowań. Składnia przekierowań zależy od shell'a jakim się posługujemy (sh, bash, tcsh).

Przekierowanie stdin i stdout (sh, bash, tcsh)

`program < plik`	przekierowanie stdin — program `program` będzie czytał z pliku `plik` zamiast z klawiatury
`program << tekst`	przekierowanie stdin przydatne w skryptach — kolejne linie skryptu aż do linii w której wystąpi tylko `tekst` będą potraktowane jako plik wejściowy
`program > plik`	przekierowanie stdout — wyniki wykonania programu `program` będą zapisane do pliku `plik` (poprzednia zawartość pliku zostanie skasowana)
`program >> plik`	przekierowanie stdout — wyniki wykonania programu `program` będą **dopisane** na końcu pliku `plik`
`program1 | program2`	przetwarzanie potokowe — podłączenie stdout programu `program1` do stdin programu `program2`

Przekierowanie stderr (sh, bash)

`program 2> plik`	przekierowanie stderr do pliku — komunikaty o błędach generowane przez `program` będą zapisane do pliku `plik` (poprzednia zawartość pliku zostanie skasowana)
`program 2>> plik`	przekierowanie stderr do pliku — komunikaty o błędach generowane przez `program` będą dopisane na końcu pliku `plik`
`program > plik1 2>plik2`	skierowanie stdout do `plik1` i stderr do `plik2`

Przekierowanie stderr (tcsh)

`program >& plik`	skierowanie stdout i stderr do `plik`
`(program > plik1) >& plik2`	skierowanie stdout do `plik1` i stderr do `plik2` — przekierowanie stderr bez przekierowania stdout jest niemożliwe

Przekierowania można łączyć ze sobą, np:

```
program1 < plik_we 2> plik_err |  program2 > plik_wy
```

`Program1` pobierze dane z pliku `plik_we`, błędy wyrzuci do pliku `plik_err`, wynik wykonania skieruje do programu `program2` który swoje wyniki wyrzuci do `plik_wy` a błędy wyświetli na ekranie (nie było przekierowania).

Przy przekierowaniach szczególnie cenne są pliki specjalne **`/dev/null`** oraz **`/dev/zero`**. Pliki te są wykorzystywane np. do ukrywania niechcianych komunikatów o błędach. Plik **`/dev/null`** przy czytaniu zachowuje się jak pusty plik, przy zapisie jest to „czarna dziura" która przyjmie dowolnie dużą ilość danych. Plik **`/dev/zero`** przy odczycie zachowuje się jak nieskończenie duży plik zawierający bajty o wartości 0, przy pisaniu zachowuje się jak **`/dev/null`**. Oba te pliki mają bardzo dużą przepustowość, znacznie większą niż dyski twarde.

Przykład 1:

Zakładamy, że w katalogu bieżącym istnieje plik o nazwie pk1 zaś nie istnieje plik pk2.

Bez rozdzielania strumieni:

```
> ls -l pk1 pk2
pk2: No such file or directory
-rw-r--r--   1 rstankie staff        193 Oct 11 13:06 pk1
```

Rozdzielenie strumieni `stdout` i `stderr`:

a) Interpreter poleceń tcsh
```
> (ls -l pk1 pk2 > out) > &error
```

b) Interpreter poleceń bash
```
> ls -l pk1 pk2 1> out 2> error
```

W obu przypadkach zostaną utworzone dwa pliki: `out` zawierający informacje o pliku `pk1` oraz `error` zawierający komunikat o błędzie.

W większości obecnych systemów istnieją pliki specjalne `/dev/random` oraz `/dev/urandom`, które zachowują się przy odczycie podobnie jak `/dev/zero`, ale zwracają losową i pseudolosową sekwencję bajtów (o szczegółach można przeczytać w linuxowym manualu w sekcji 4 pod nazwą random: man 4 random). Odczyt z tych plików jest jednak wielokrotnie niższy niż w przypadku `/dev/zero` i bardzo silnie obciąża procesor.

3. Polecenie cat

Często zachodzi potrzeba wyświetlenia jednego pliku lub połączonej zawartości kilku plików bez wprowadzania w nich jakichkolwiek modyfilkacji. Do takich zadań służy polecenie **`cat`**.

```
cat plik1 plik2 plik3
```

wyświetli na ekranie połączoną zawartość plików `plik1 plik2 plik3`. Oczywiście można użyć przekierowania: `cat plik1 plik2 plik3 > plik4`

Polecenie `cat` jest również często używane do wyświetlania zawartości krótkich plików tekstowych.

4. Polecenie dd

To polecenie służy do kopiowania danych blokami o określonej długości. Z tego powodu bywa często używane przy operacjach na plikach będących urządzeniami blokowymi (np. dysk twardy, partycja, stacja dyskietek).

Podstawowa składnia polecenia:

dd bs=1k count=100 if=plik1 of=plik2 skip=1 seek=2

bs	block size, domyślnie 512 bajtów
count	ile bloków ma być skopiowane, domyślnie wszystkie
if	plik wejściowy, domyślnie stdin
of	plik wyjściowy, domyślnie stdout
skip	ile bloków pominąć w pliku wejściowym (domyślnie 0)
seek	od którego bloku zacząć zapis w pliku/urządzeniu wyjściowym (domyślnie 0)

Przykład 2

Utworzenie pliku test o rozmiarze 1024kB wypełnionego bajtami o wartości 0

```
$ dd bs=1k count=1024 if=/dev/zero of=test
1024+0 records in
1024+0 records out
$ ls -l test
-rw-r--r--    1 szyman   staff     1048576 2004-10-13 11:23 test
```

5. Pliki „sparse”

Są to pliki w których występują „dziury” w danych. Przy odczycie miejsca w których występują dziury wyglądają tak, jakby były tam zapisane bajty o wartości 0, jednakże „dziura” nie zajmuje miejsca na dysku. W miarę zapisywania danych w obszarze „dziury” plik zajmuje coraz więcej miejsca na dysku bez widocznej zmiany swojego rozmiaru.

Rozmiar wirtualny pliku typu sparse jest widoczny w wyniku działania komendy ls -l, rozmiar rzeczywisty poda komenda du (disk usage). Plik typu sparse powstaje automatycznie jeśli zażądamy zapisu danych na pozycji dalszej niż aktualny koniec pliku. Od aktualnego końca pliku aż do pozycji w której zaczynają się zapisane dane powstaje „dziura".

Na pliki typu sparse należy uważać przy kopiowaniu danych - nie każda komenda potrafi skopiować plik sparse tak, żeby plik docelowy był również plikiem sparse. Pokazuje to przykład:

Przykład 3

Utworzenie pliku sparse o rozmiarze wirtualnym 1GB, du -k podaje rozmiar w blokach 1kB

```
$ dd bs=1024k count=1 if=/dev/zero of=test seek=1024
1+0 records in
1+0 records out
$ ls -l test
-rw-r--r--    1 szyman   staff    1074790400 2004-10-13 11:46 test
$ du -k test
1056    test
```

Komenda cp wykrywa długie ciągi bajtów o wartości zero i sama zastępuje je „dziurami", zastąpiła nawet ostatnie 1024kB faktycznie zapisanych danych.

```
$ cp test test1
$ ls -l test1
-rw-r--r--    1 szyman   staff    1074790400 2004-10-13 11:48 test1
$ du -k test1
16      test1
```

Teraz plik typu sparse nie powstanie, zaś na plik „standardowy" na dysku braknie miejsca (było ok. 200MB wolnego miejsca)

```
$ rm test1
$ cat test > test1
cat: write error: No space left on device
```

6. Polecenia tar, gzip, bzip2

Polecenie tar służy do tworzenia archiwów plików zbliżonych w sposobie działania do archiwów tworzonych programami zip albo rar. Co prawda program zip jest dostępny na większości systemów unixowych ale format jakim się posługuje nie jest dostosowany do specyfiki unixa, w szczególności zip nie zapisuje praw dostępu oraz informacji o właścicielu i grupie pliku. Archiwa tworzone tar'em nie mają tych mankamentów, i są de facto standardem w świecie unix'a.

Tworzenie archiwum `archiwum.tar` z katalogów `katalog1 katalog2` itd.:
`tar cf archiwum.tar katalog1 katalog2 ...`

Testowanie archiwum **`archiwum.tar`**
`tar tf archiwum.tar`

Rozpakowanie archiwum **`archiwum.tar`** w bieżącym katalogu
`tar xf archiwum.tar`

Archiwa tworzone tar'em nie są skompresowane. Do kompresji tych archiwów jest powszechnie używany program gzip, czasm bzip2. Wynikowe rozszerzenie archiwów po kompresjii to .tar.gz (czasem .tgz) albo .tar.bz2 . Spakowanie i rozpakowanie archiwum z użyciem gzip'a jest wykonywane zwykle z użyciem potoku (znak minus po opcji f w poleceniu tar oznacza zapisz archiwum do stdout albo czytaj archiwum z stdin):

```
tar cf - katalog1 katalog2 | gzip -c > archiwum.tar.gz
gzip -dc archiwum.tar.gz | tar tf -

tar cf - katalog1 katalog2 | bzip2 -c > archiwum.tar.bz2
bzip2 -dc archiwum.tar.bz2 | tar tf -
```

W wersji GNU tar'a istnieją dodatkowe opcje upraszczające kompresję archiwów. Użycie opcji **z** oznacza, że utworzone archiwum będzie od razu skompresowane gzip'em, opcja **j** oznacza kompresję za pomocą bzip2.

```
tar czf archiwum.tar.gz katalog1 katalog2 ...
tar tzf archiwum.tar.gz
tar xzf archiwum.tar.gz

tar cjf archiwum.tar.bz2 katalog1 katalog2 ...
tar tjf archiwum.tar.bz2
tar xjf archiwum.tar.bz2
```

7. Polecenie md5sum

To polecenie pozwala obliczyć sumę kontrolną md5 dla podanego pliku (można podać nazwę pliku albo użyć stdin) i bywa często używane do weryfikacji, czy plik pobrany z sieci jest poprawny (autor zamieszcza plik oraz jego sumę kontrolną w pliku z rozszerzeniem .md5)

$ md5sum /etc/passwd
dbc5e89c489d53a52f6a34b54be2a32c /etc/passwd

8. Pliki specjalne

Pliki specjalne reprezentują urządzenia, takie jak dysk, partycja, napęd taśm, cdrom, karta muzyczna, port COM, oraz urządzenia wirtualne - takie jak generator liczb losowych, „czarna dziura" czy generator bajtów o wartości zero. Taka reprezentacja ułatwia korzystanie z tych urządzeń gdyż pozwala na wykonanie pewnych operacji na nich za pomocą standardowych funkcji i poleceń.

I tak dla pliku specjalnego /dev/fd0 reprezentującego stację dyskietek komenda:

```
dd if=/dev/fd0 of=floppy.img
```

spowoduje utworzenie pliku floppy.img zawierającego obraz (wierną kopię) dyskietki, komenda

```
dd if=floppy.img of=/dev/fd0
```

nagra ten obraz z powrotem na dyskietkę. W ten sposób można pod unixem skopiować dyskietkę.

Komenda `dd if=/dev/zero of=/dev/fd0` zapisze dyskietkę zerami kasując wszystkie zapisane tam dane (łącznie ze strukturami katalogów itp.) Odzyskanie tak skasowanych danych jest możliwe jedynie w specjalizowanych laboratoriach na podstawie pozostałości magnetycznych (pod mikroskopem).

Urządzenia blokowe (zapisujące dane w blokach, np. dyskietka, dysk twardy) są reprezentowane przez pliki specjalne blokowe:

```
$ ls -l /dev/fd0
brw-rw----    1 root     floppy     2,   0 Sep 15  2003 /dev/fd0
```

Urządzenia znakowe (zapisujące strumień danych bez możliwości wykonania operacji typu cofnij, idź do bajtu) są reprezentowane przez pliki specjalne znakowe:

```
$ ls -l /dev/zero
crw-rw-rw-    1 root     root       1,   5 Sep 15  2003 /dev/zero
```

Każde urządzenie jest jednoznacznie reprezentowane przez trójkę (typ, major device id, minor device id) gdzie typ to urządzenie blokowe – b, albo znakowe – c. Przypisanie różnych kombinacji typ, minor, major do poszczególnych urządzeń jest standardowe (takie samo w różnych wersjach i dystrybucjach systemów z rodziny Unixa). Urządzenie nie jest natomiast ściśle związane z nazwą pliku specjalnego. Zwyczajowo przyjęto jednak pewne nazwy. Np. /dev/hda1 to pierwsza partycja na pierwszym dysku twardym, /dev/fd0 to stacja dysków itp.

```
crw-rw-rw-    1 root     root       1,   5 Sep 15  2003 /dev/zero
                                    ^^^^  ^^^
                                   major  minor
                                  dev id  dev id
```

Przykładowo, utworzenie pliku specjalnego blokowego /tmp/aa o parametrach major=2 minor=0 spowoduje, że ten plik będzie również reprezentował stację dyskietek (tak jak /dev/fd0):

```
$ mknod /tmp/aa b 2 0
$ ls -l /tmp/aa
brw-rw-rw-    1 root     floppy     2,   0 Oct 13  2003 /tmp/aa
```

Prawa dostępu na pliku specjalnym decydują o tym, kto może korzystać bezpośrednio ze wskazanego urządzenia (czyli np. sformatować dyskietkę).

9. Pytania kontrolne

1. Jak wyjść z edytora vi
2. Mamy plik dane.tar . W jaki sposób utworzyć plik dane.tar.md5 zawierający sumę kontrolną md5 pliku dane.tar
3. W jaki sposób można przekierować wynik działania komendy ls -l do pliku aa ?
4. W jaki sposób można spowodować, że komenda wyświetla na ekranie tylko komunikaty o błędach?
5. Jak zachowuje się plik /dev/zero przy odczycie
6. Jak zachowuje się plik /dev/null przy zapisie
7. Co to jest plik typu sparse?
8. Mając dwa pliki specjalne blokowe, jak stwierdzić, czy reprezentują to samo urządzenie?
9. Jakie trzeba mieć uprawnienia, żeby sformatować dyskietkę w systemie Unix?
10. Podaj nazwę komendy jakiej trzeba użyć aby stworzyć obraz dysku.
11. W jaki sposób można stwierdzić, czy dwa pliki posiadają jednakową zawartość?
12. Do czego służy przekierowanie <<
13. Czym się różni przekierowanie > od >>
14. Jaka komenda rozpakuje plik dane.tar w bieżącym katalogu?

10. Ćwiczenia

Uruchom maszynę wirtualną AKISO/SO , zaloguj się jako student i otwórz okno terminala.

Ćwiczenie 1:
Polecenie **`du -sk *`** wyświetla informację o rozmiarach (w kB) plików i katalogów (z podkatalogami) znajdujących się w bieżącym katalogu.

Będąc w katalogu /etc i działając z konta użytkownika student (nie root !) zapisz wynik działania tego polecenia do pliku /tmp/rozmiary

Zauważ, że informacja o braku prawa dostępu do niektórych katalogów jest nadal wyświetlana na ekranie.

Wykonaj jeszcze raz to ćwiczenie tak, aby komunikaty o błędach nie wyświetlały się na ekranie i nie były nigdzie zapisane.

Ćwiczenie 2:
W poprzednim ćwiczeniu uzyskano listę rozmiarów plików. Chcąc dowiedzieć się które pliki lub katalogi zajmują najwięcej miejsca należy tę listę posortować w porządku numerycznym. Do tego celu może służyć polecenie sort -n, które pobiera dane do sortowania z stdin i wyrzuca je posortowane (opcja -n zapewnia sortowanie w porządku numerycznym) na stdout.

Korzystając z przekierowań zapisz posortowaną zawartość pliku /tmp/rozmiary do pliku /tmp/rozmiary.sorted

Wyświetl zawartość pliku /tmp/rozmiary.sorted

Ćwiczenie 3:
Uzyskanie informacji w ćwiczeniach 1 i 2 wymagało zapisu dwóch plików z pośrednimi danymi. Wynik (posortowaną listę rozmiarów plików) można również uzyskać bez tworzenia plików pośrednich, przetwarzając dane w potoku. Jest to szczególnie cenne, gdy pliki pośrednie mają bardzo duże rozmiary i nie mieszczą się na dysku.

Wyświetl jednym poleceniem (składającym się z kilku komend połączonych potokami) posortowaną numerycznie listę rozmiarów plików i katalogów znajdujących się w katalogu /etc . Pamiętaj o usunięciu komunikatów o błędach i o zastosowaniu polecenia less do wyświetlania danych po 1 stronie.

Ćwiczenie 4:
W tym ćwiczeniu pobierzesz z sieci archiwum ze źródłami narzędzi do obsługi Logical Volume Manager w systemie linux. Operacje, które wykonasz na tym archiwum należą do powszechnie wykonywanych czynności przy instalacji oprogramowania oraz przy przenoszeniu danych pomiędzy unix'ami.

Jako użytkownik student za pomocą polecenia
`wget ftp://sources.redhat.com/pub/lvm/current/lvm_1.0.8.tar.gz`
ściągnij plik lvm_1.0.8.tar.gz do bieżącego katalogu. Pobierz także plik md5.sum zawierający sumy kontrolne plików (`wget ftp://sources.redhat.com/pub/lvm/current/md5.sum`).

Sprawdź, czy suma kontrolna pliku lvm_1.0.8.tar.gz zgadza się z sumą kontrolną zapisaną w pliku md5.sum (ten plik zawiera sumy kontrolne dla większej liczby plików).

Przetestuj archiwum lvm_1.0.8.tar.gz korzystając z odpowiednich opcji komendy tar. Spróbuj się zorientować czy rozpakowanie archiwum utworzy katalog z plikami, czy wrzuci pliki z archiwum „luzem" do bieżącego katalogu. Upewnij się także, że ścieżki w archiwum nie zaczynają się od znaku / co oznacza ścieżki bezwzględne i grozi rozpakowaniem plików w różnych (niechcianych) miejscach systemu plików.

Rozpakuj archiwum w bieżącym katalogu. Powinien powstać nowy katalog o nazwie LVM. Wyświetl zawartość tego katalogu, sprawdź, kto jest właścicielem plików (wejdź głębiej w strukturę tego katalogu).

Otwórz konsolę root'a i rozpakuj ponownie ściągnięte archiwum (nie ściągaj go ponownie). Podobnie jak poprzednio powinien powstać katalog LVM. Kto jest teraz właścicielem podkatalogów i plików w katalogu LVM? Zapamiętaj tę różnicę w działaniu tar'a z konta zwykłego użytkownika i z konta root'a.

Ćwiczenie 5:
Utwórz nowe archiwum o nazwie LVM.tar.gz zawierające katalog LVM. Przetestuj je.

Użyj teraz do testowania opcji **`d`** zamiast opcji **`t`** — to spowoduje porównanie zawartości archiwum z dyskiem (nie powinno być różnic). Zmień zawartość dowolnego pliku w katalogu LVM. Ponownie przetestuj archiwum z użyciem opcji **`d`**.

Ćwiczenie 6:
Za pomocą komendy:
```
split -b 150000 lvm_1.0.8.tar.gz part.
```

podziel oryginalne archiwum na pliki part.aa, part.ab, part.ac o długości 150000 bajtów.

Za pomocą komendy
```
cat part.* | md5sum
```

złóż oryginalne archiwum z części i oblicz sumę kontrolną md5. Czy suma md5 zgadza się z sumą dla oryginalnego archiwum?

W powyższy sposób możesz podzielić dowolny plik na mniejsze części (dopasowane do rozmiaru dyskietki, płyty CD lub innego nośnika) i złożyć go z powrotem, oraz przetestować, czy plik został złożony poprawnie.

Złóż części archiwum w złej kolejności i oblicz sumę md5:
```
cat part.ac part.ab part.aa | md5sum
```

Czy suma kontrolna się zgadza?

Ćwiczenie 7:
Usuń katalog LVM.
Jednym poleceniem połącz i rozpakuj archiwum zawarte w plikach part.aa ... part.ac.

Ćwiczenie 8:
Warto wiedzieć, jak szybko utworzyć plik o określonym rozmiarze, wypełniony zerami. Jest to szczególnie przydatne przy testach (np. szybkości zapisu/odczytu/transmisji w sieci) oraz przy tworzeniu obrazów dysków.

Utwórz plik o rozmiarze 2kB wypełniony bajtami o wartości 0.

Ćwiczenie 9:
Wykonaj polecenia z Przykładu 3 zamieszczonego w materiałach, ale utwórz plik sparse o rozmiarze 2.5GB (na dysku jest ok. 2GB wolnego miejsca, chodzi o to żeby go zabrakło).

Ćwiczenie 10:
Za pomocą komendy ls -l wyświetl informację o pliku specjalnym /dev/zero. Jaki jest typ tego urządzenia? Jakie są identyfikatory major, minor tego urządzenia?

Z konsoli roota:
usuń plik /dev/zero , następnie utwórz go ponownie za pomocą komendy:
```
mknod /dev/zero type major minor
```

gdzie:
`type` - litera b albo c w zależności od typu urządzenia (b - blokowe, c - znakowe)
`major` - identyfikator major device id
`minor` - identyfikator minor device id

Utwórz plik /tmp/aaa, który będzie reprezentował to samo urządzenie co /dev/zero .
Spróbuj przeczytać z utworzonego pliku kilanaście GB danych i sprawdź, czy są to bajty o wartości 0.

Laboratorium 3

- **System plików — urządzenia blokowe**

--- Ostrzeżenie !!! ---

Przedstawiony tekst zawiera komendy, które **mogą spowodować utratę danych** zapisanych na dyskach twardych fizycznie przyłączonych do komputera. **Dotyczy to również sytuacji, w której komendy są wydane w systemie Linux uruchomionym z CD!!!**

1. Dyskietki

Jeszcze nie dawno w powszechnym użyciu były dyskietki o rozmiarze 3,5 cala o pojemności 1.44MB. Dane na dyskietce są zapisywane w sektorach o rozmiarze 512 bajtów, rozmieszczonych na okręgach, zwanych ścieżkami. Do zapisu są używane 2 głowice, po jednej dla każdej strony dyskietki. Zbiór ścieżek o tym samym promieniu, obsługiwanych przez różne głowice jest nazywany cylindrem.

Format dyskietki 1.44MB to: 80 cylindrów, 2 głowice, 18 sektorów na ścieżkę, 512 bajtów w sektorze, co daje: 80×2×18×512=1474560 bajtów (1440KB, przy założeniu, że 1KB to 1024 bajty).

Z punktu widzenia unix'a dyskietka jest urządzeniem, zawierającym 2880 sektorów o rozmiarze 512 bajtów, bez wyróżniania ścieżek, cylindrów itp. Zerowy sektor dyskietki jest sektorem startowym, który może zawierać program ładujący system operacyjny (w przypadku dyskietki startowej systemu operacyjnego). Dyskietka zawiera jeden system plików (nie jest dzielona na mniejsze części). W systemie linux pierwsza stacja dyskietek jest reprezentowana przez plik /dev/fd0.

2. Dysk twardy

Dane na dysku twardym, podobnie jak na dyskietce, są zapisywane w sektorach o rozmiarze 512 bajtów. Pierwsze dyski twarde miały stałą liczbę sektorów na ścieżce, a sektor na dysku był identyfikowany przez trójkę CHS (cylinder, head, sector). W tych dyskach geometria fizyczna dysku (liczba cylindrów, głowic, sektorów na ścieżce) była znana i używana bezpośrednio przez system operacyjny.

W obecnych dyskach twardych ścieżki zawierają różną liczbę sektorów - zewnętrzne więcej, wewnętrzne mniej, co ma na celu efektywne wykorzystanie nośnika. Takie dyski prezentują systemowi operacyjnemu geometrię logiczną, w której pojemność dysku jest zbliżona do pojemności rzeczywistej, a parametry CHS zależą od zastosowanej translacji i zwykle nie mają nic wspólnego z rzeczywistością (np. 255 głowic). Ubocznym efektem zapisu z różną liczbą sektorów na ścieżce jest zdecydowanie większa (zwykle dwukrotnie) szybkość transferu danych z początku dysku (ścieżki zewnętrzne) w stosunku do szybkości transferu z końca dysku (ścieżki wewnętrzne). Warto brać to pod uwagę testując szybkość dysku oraz planując rozmieszczenie danych na dyskach.

Adresacja CHS niesie ze sobą różne ograniczenia pojemności (dyskusję tych ograniczeń zawiera dokument http://www.tldp.org/HOWTO/Large-Disk-HOWTO.html). Równolegle z nią jest stosowana adresacja LBA (Logical Block Addressing), w której sektory na dysku są ponumerowane kolejno od 0. Linux korzysta w zasadzie wyłącznie z adresacji LBA.

W zależności od zastosowanego interfejsu możemy mieć do czynienia z dyskami ATA, Serial ATA (SATA), SCSI, SAS (Serial Attached SCSI) oraz FC (Fibre Channel).

Dyski SCSI, SAS, FC oraz SATA są widoczne w linux'ie jako urządzenia /dev/sdX (/dev/sda, /dev/sdb, itd.)
Urządzenia ATA są widoczne jako /dev/hdX (/dev/hda, /dev/hdb, itd.) choć zdarza się, że niektóre kontrolery prezentują dyski ATA używając notacji /dev/sdX.

Napędy CD-ROM SCSI są widoczne jako /dev/scd0, /dev/scd1 itd, albo /dev/sr0, /dev/sr1, itd.
Napędy CD-ROM ATA są widoczne tak jak dyski ATA (/dev/hdX)

Zwykle pierwszy napęd CD jest dodatkowo dostępny przez plik /dev/cdrom.

W przypadku urządzeń ATA możliwe jest podłączenie do dwóch urządzeń ATA na jednym kanale (porcie) kontrolera. Jedno z tych urządzeń musi być skonfigurowane jako MASTER drugie jako SLAVE. Przypisanie oznaczeń do urządzeń jest następujące (nie ma znaczenia, czy wszystkie urządzenia są podłączone):

- /dev/hda - Kanał 1, MASTER (Primary Master)
- /dev/hdb - Kanał 1, SLAVE (Primary Slave)
- /dev/hdc - Kanał 2, MASTER (Secondary Master)
- /dev/hdd - Kanał 2, SLAVE (Secondary Slave)

W przypadku, gdy w komputerze jest więcej kanałów ATA, dalsze urządzenia są oznaczane kolejnymi literami.

Dyski twarde mogą być wykorzystywane bezpośrednio, jednakże najczęściej są one dzielone na partycje. Istnieje wiele systemów podziału na partycje. W architekturze opartej na Intel x86 i pochodnych najpopularniejszym systemem opisu partycji jest tablica partycji DOS (DOS partition table), używana przez systemy DOS, Windows i często Linux oraz rozszerzenie wprowadzone niedawno przez Microsoft (tzw. dyski dynamiczne - Windows XP/Server 2003). Spotyka się także opis partycji w formacie Sun Disklabel używanej przez systemy Solaris a także (często) FreeBSD. Stosunkowo nowym formatem tablicy partycji jest GPT (GUID Partition Table) zaproponowana przez firmę Intel. Format ten usuwa ograniczenia starego formatu DOS Partition Table, w szczególności limitu 2TB na rozmiar partycji, i może być używany w najnowszych systemach Windows, Linux oraz Solaris.

Tablica partycji DOS jest umieszczona w zerowym sektorze dysku twardego (Master Boot Record – MBR). MBR zawiera:

- Kod startowy który jest wykonywany przy uruchomieniu komputera (446 bajtów), przy czym Windows 2000/XP w ostatnich 6 bajtach tego obszaru zapisują unikalną sygnaturę dysku,
- Tablicę partycji:
  - 16 bajtów opisu pierwszej partycji podstawowej
  - 16 bajtów opisu drugiej partycji podstawowej
  - 16 bajtów opisu trzeciej partycji podstawowej
  - 16 bajtów opisu czwartej partycji podstawowej
  - 2 bajty tzw. magic number (zawsze ustawione na 0x55, 0xAA, jest to najprostszy test na to, że sektor zawiera poprawną tablicę partycji)

Z przedstawionego opisu wynika, że na dysku z tablicą partycji DOS mogą być maksymalnie 4 partycje podstawowe. Jedna z tych partycji może być partycją rozszerzoną, zawierającą w sobie dowolną liczbę dysków (partycji) logicznych.

Dla dysku /dev/hda partycje podstawowe mają oznaczenia /dev/hda1, /dev/hda2, /dev/hda3, /dev/hda4. Pierwszy dysk (partycja) logiczna ma zawsze oznaczenie /dev/hda5, kolejne /dev/hda6, /dev/hda7, itd.. Podobnie dla /dev/hdb i dalszych oraz /dev/sda i dalszych.

16 bajtów przeznaczonych na opis partcji zawiera:

- znacznik „boot" - oznacza tzw. partycję aktywną, z którj należy załadować system operacyjny
- informacje o początku partycji
- informacje o długości/końcu partycji
- typ partycji

Typ partycji określa jej zawartość. Dla linuxa najważniejsze typy partycji to:

- 0x05 - extended (partycja rozszerzona)
- 0x82 - linux swap
- 0x83 - linux
- 0x8e - linux LVM
- 0xfd - linux RAID autodetect

Do podziału na partycje służą pod linux'em narzędzia `fdisk`, `sfdisk`, `cfdisk`, różniące się funkcjonalnością i sposobem komunikacji z użytkownikiem.

Polecenie `fdisk` uruchamia się z podaniem dysku do partycjonowania, np. `fdisk /dev/hda`
Dalsze operacje wykonuje się interaktywnie podając jednoliterowe komendy.

Fundamentalną regułą przy tworzenia partycji jest aby partycje były rozłączne (nie zachodziły na siebie). Między partycjami mogą występować przerwy. Wskazane jest (ze względu na zgodność z Windows) aby partycje były ułożone kolejno na dysku (tzn.w kolejności hda1, hda2 itd).

Rozpoczynając modyfikację układu partycji na dysku należy zwrócić uwagę, aby modyfikowane partycje nie były używane przez system operacyjny. **Kończąc pracę z fdisk'iem należy zwrócić uwagę, czy modyfikacja tablicy partycji została uwzględniona przez jądro systemu natychmiast czy też nie:**

Komunikat:

```
WARNING: Re-reading the partition table failed with error 16: Device or resource busy.
The kernel still uses the old table.
The new table will be used at the next reboot.
```

oznacza, że jedna z partycji na w/w dysku była używana. W takiej sytuacji nowy układ dysku zostanie uwzględniony dopiero po restarcie komputera. Można także usunąć przyczynę „zajętości" tablicy partycji i wymusić ponowne odczytanie tablicy partycji przez jądro systemu, np. ponownie uruchamiając fdisk i wykonując jeszcze raz zapis tablicy partycji na dysk.

Najprostszym sposobem usunięcia tablicy partycji jest zapis pierwszego sektora dysku zerami, np.:
`dd count=1 if=/dev/zero of=/dev/hda`

w podobny sposób można wyczyścić cały dysk.

Można również wyczyścić pojedynczą partycję:
`dd if=/dev/zero of=/dev/hda1`

taka operacja usuwa wszystkie dane z partycji /dev/hda1 nie naruszając danych zapisanych na innych partycjach (Uwaga - łatwo tutaj o tragiczną w skutkach pomyłkę).

Pierwszy sektor każdej partycji może, podobnie jak dyskietka, zawierać kod startowy systemu operacyjnego.

Obecne dyski twarde dysponują mechanizmem podmiany uszkodzonych sektorów na zapasowe, normalnie nie wykorzystywane i nie wliczane do pojemności dysku. Podmiana jest realizowana w całości przez dysk i jest niewidoczna dla systemu operacyjnego. Jeżeli uszkodzenie zostanie wykryte przy zapisie, to uszkodzony sektor jest podmieniany natychmiast. Jeżeli uszkodzenie zostanie wykryte przy odczycie, dysk spróbuje odczytać uszkodzony sektor wiele razy (towarzyszą temu charakterystyczne efekty dźwiękowe). Jeżeli sektor zostanie w końcu odczytany poprawnie to dane zostaną natychmiast skopiowane do sektora zapasowego i nastąpi podmiana sektora. Jeżeli sektor nie da się odczytać, dysk będzie wykazywał błędy odczytu z tego sektora. Podmiana na sektor zapasowy nastąpi przy najbliższym zapisie do sektora uszkodzonego.

W celu umożliwienia monitorowania parametrów niezawodnościowych dysków wyposażono je w technologię SMART (Self Monitoring Analysis and Reporting Technology). Za pomocą odpowiednich narzędzi (np. `smartctl` w linux'ie) jest możliwa obserwacja parametrów dysku, takich jak liczba uszkodzonych sektorów, temperatura dysku, liczba godzin pracy. Możliwe jest również przeglądanie rejestru zdarzeń oraz zlecanie niskopoziomowych testów.

Polecenie:
`smartctl -a /dev/hda`

wyświetli wszystkie parametry SMART związane z dyskiem /dev/hda.

3. Systemy plików

Najprostszymi systemami plików są systemy rodziny FAT (FAT12, FAT16, FAT32). Jednostką przestrzeni dyskowej w tych systemach jest klaster (blok) o rozmiarze będącym wielokrotnością rozmiaru sektora dysku, jego rozmiar zależy od wielkości woluminu (partycji, dyskietki) i od zastosowanego wariantu systemu FAT. Każdy plik zajmuje jeden, lub więcej klastrów

Adres klastra jest zapisywany na 12, 16 bądź 32 bitach, co pozwala na zaadresowanie odpowiednio 4096, 65536 oraz 4294967296 klastrów. System FAT12 jest używany w zasadzie wyłącznie na dyskietkach, ze względu na

bardzo małą liczbę dostępnych klastrów. Z kolei łatwo policzyć, że np. FAT16 dla partycji o rozmiarze bliskim 1GB wymaga stosowania klastrów o rozmiarze 16KB (nieefektywne wykorzystanie przestrzeni dyskowej), co wyklucza jego stosowanie dla dużych partycji.

Charakterystyczną cechą systemów FAT jest to, że posługują się one tablicą alokacji plików (stąd nazwa - File Allocation Table), która przechowuje informacje o klastrach należących do poszczególnych plików. Wpis w katalogu zawiera informację o pierwszym klastrze pliku, numer następnego klastra można odczytać w tablicy FAT pod indeksem poprzedniego klastra pliku. Takie rozwiązanie jest poważnym ograniczeniem wydajności przy dużych systemach plików.

System FAT nie obsługuje praw dostępu, nie posługuje się także pojęciem i-node, stąd nie jest możliwe tworzenie w tym systemie dowiązań miękkich i twardych. Nie jest także możliwe tworzenie plików specjalnych. Oryginalnie systemy FAT16 i FAT12 obsługiwały nazwy plików w postaci 8 znaków nazwy i 3 znaków rozszerzenia. Stosowana później modyfikacja pozwoliła na korzystanie z dłuższych nazw.

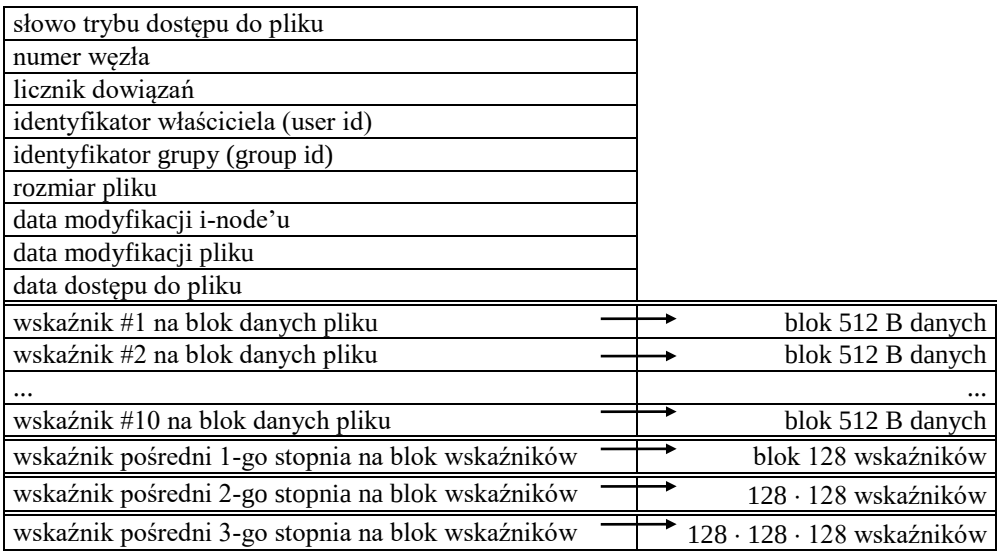
Systemy FAT są bardzo podatne na błędy spowodowane niedokończonym zapisem (np. w wyniku awarii zasilania bądź zresetowania komputera w czasie zapisu). Nie ma tutaj nawet mechanizmu pozwalającego na stwierdzenie, że taka sytuacja miała miejsce. Typowym błędem jest zagubienie klastra (lost cluster), w którym jeden z klastrów jest raportowany jako zajęty, ale nie należy do żadnego pliku. Konieczne jest więc regularne sprawdzanie dysku za pomocą chkdsk (Windows) albo fsck (Unix).

Zaletą systemu FAT jest jego powszechność - praktycznie każdy system operacyjny potrafi go obsłużyć.

W systemie Linux powszechnie stosowanym systemem plików jest ext2 (oraz jego nowsza odmiana ext3) wywodzący się z systemu plików minix. Konstrukcja tego systemu plików jest typowa dla systemów Unix. Rozmiar bloku (klastra) jest ustalany w momencie tworzenia systemu plików, typowo jest to od 512 bajtów do 4KB (wielokrotność rozmiaru sektora dysku). Rozmiar bloku wpływa na efektywność przydziału miejsca i szybkość pracy. Mniejsze bloki dają efektywny przydział miejsca, ale odczytanie pliku wymaga odczytania większej liczby bloków, które mogą być rozrzucone na dysku.

Każdy plik jest reprezentowany przez strukturę informacyjną nazywaną węzłem pliku (i-node). Każdy plik jest jednoznacznie identyfikowany przez węzeł. W jednym systemie plików nie ma dwóch plików o tym samym numerze węzła. Zasadniczo przez „plik" będziemy więc rozumieć konkretny węzeł. Przez utworzenie nowego pliku rozumiemy utworzenie nowego węzła.

Struktura węzła jest następująca:



Przy blokach 512 bajtowych maksymalny rozmiar pliku w UNIX wynosi więc 1 GB, a przy blokach o wielkości 1 kB wynosi 16 GB. Jak łatwo zauważyć w węźle pliku nie jest przechowywana jego nazwa. Powiązanie nazw z numerami węzłów jest przechowywane w plikach-katalogach.
Położenie węzła (i-node'u) katalogu głównego (root directory) danego systemu plików jest ściśle określone. Jest to węzeł o numerze 2.

System ext2 jest również podatny na błędy powstałe w wyniku niekompletnego zapisu, ale dysponuje znacznikiem, pozwalającym na stwierdzenie, czy taka sytuacja miała miejsce. Jeżeli tak - to wymagane jest uruchomienie programu fsck który naprawi błędy. Zwykle jest to wykonywane automatycznie przy starcie systemu. Sprawdzanie woluminu jest wykonywane również wtedy, gdy zostanie przekroczona określona liczba podłączeń systemu plików, albo gdy od ostatniego sprawdzania minie określony czas (obie wartości można ustawiać).

Przy tworzeniu woluminu w systemie ext2 ważne jest właściwe określenie maksymalnej liczby węzłów (i-node), gdyż liczba ta nie może być zmieniona bez ponownego utworzenia systemu plików. Z jednej strony deklaracja zbyt dużej liczby węzłów powoduje stratę miejsca na dysku, z drugiej strony przy zadeklarowaniu zbyt małej liczby węzłów może ich braknąć (co uniemożliwi tworzenie nowych plików mimo, że na woluminie jest nadal wolne miejsce).

Warto wiedzieć, że w systemie ext2 część bloków jest zarezerwowana do użycia przez root'a (lub innego wskazanego użytkownika) w razie wyczerpania się wolnego miejsca na dysku.

ext2 obsługuje quotę - czyli limit miejsca dostępnego dla użytkownika albo grupy. Limitu twardego (hard) przekroczyć nie można. Limit miękki (soft limit) to taki, który można przekroczyć, ale tylko przez pewien ustalony okres czasu (grace period), po którym zapis na wolumin jest blokowany aż do momentu w którym ilość zajętego miejsca spadnie poniżej limitu. Limity obejmują liczbę zajętych bloków na dysku (klastrów) oraz liczbę wykorzystanych węzłów.

Szczegółowe informacje na temat systemu plików ext2 można zdobyć ze strony:
http://e2fsprogs.sourceforge.net/ext2intro.html

ext3 jest rozszerzeniem ext2 i należy do rodziny systemów plików z dziennikiem (journalling filesystems), do której należą również NTFS (Windows NT/2000/XP), XFS (Silicon Graphics, Linux), UFS (Solaris).

W tych systemach dane użytkownika są obsługiwane normalnie, ale wszystkie zmiany w strukturach systemu plików (tablice węzłów, tablice zajętości bloków, itp.) są wykonywane dwuetapowo:

- zapis zmian do pliku dziennika
- wykonanie operacji z dziennika na strukturach systemu plików

Dzięki takiej organizacji awaria systemu (utrata zasilania, reset) przed ukończeniem zapisu do dziennika powoduje, że system plików jest nieuszkodzony (w wersji sprzed zmian) a uszkodzone wpisy w dzienniku są ignorowane przy następnym uruchomieniu systemu. Jeżeli awaria wystąpi w drugiej fazie zapisu, przy uruchomieniu systemu wystarczy jeszcze raz wykonać operacje zapisane w dzienniku.

Takie rozwiązanie wybitnie skraca czas potrzebny na przywrócenie systemu plików do normalnego stanu po awarii.

Oprócz opisanych systemów plików istnieją systemy plików o innej organizacji danych wewnątrz systemu plików. Jednym z takich systemów jest reiserfs (www.namesys.com), stosujący organizację danych w postaci drzew binarnych, co pozwala na bardzo szybkie wyszukiwanie plików. Innymi przykładami jest system plików iso9660, stosowany przy zapisie dysków CD-R oraz CD-RW, oraz UDF (Universal Disk Format) i MRW (Mount Rainer) stosowane przy zapisie dysków CD-RW.

4. Obsługa woluminów

System plików na woluminie tworzymy za pomocą komendy `mkfs`, podając typ systemu plików oraz inne informacje potrzebne do utworzenia systemu plików np.:

```
mkfs -t ext3 /dev/hda1
```

jest to odpowiednik komendy `format` w systemach DOS/Windows.

Sprawdzenie woluminu jest wykonywane komendą fsck, np.:

```
fsck /dev/hda2
```

Inne operacje, takie jak wyświetlanie i ustawianie parametrów systemu plików wykonywane są komendami dedykowanymi dla konkretnego systemu plików, przykładowo dla systemów ext2/ext3 są to narzędzia `dumpe2fs` oraz `tunefs`.

Wolumin jest przyłączany do istniejącego drzewa systemu plików w miejscu wskazanego katalogu. Katalog ten musi wcześniej istnieć, nie zawierać plików (jeśli zawiera on jakieś pliki to stają się one niedostępne aż do czasu odłączenia woluminu, choć istnieją wyjątki od tej reguły).

Do przyłączania woluminów służy komenda `mount`, np.:

```
mount /dev/hda1 /home
```

Aktualnie podłączone woluminy można wyświetlić komendą `mount` bez argumentów.
Komenda `mount` akceptuje różne opcje, część z nich jest wspólna dla wszystkich typów systemów plików (np. ro - read only, rw - read/write), część jest specyficzna dla konkretnego typu systemu plików.

Do odłączania woluminów służy komenda umount, np.:
```
umount /dev/hda1
umount /home
```

Odłączenie woluminu jest możliwe tylko wtedy, gdy żaden proces z niego nie korzysta, czyli nie ma w nim otwartych plików i nie ma ustawionego bieżącego katalogu w tym woluminie.

Aby się dowiedzieć, które procesy korzystają z woluminu należy użyć komendy `lsof`, np:
```
lsof /home
```

W wielu systemach unix jest stosowany automounter, który zajmuje się automatycznym podłączaniem dyskietek, płyt cd i katalogów sieciowych w momencie ich użycia.

Ilość miejsca na podłączonych woluminach podaje komenda `df` (`df -k` poda informację w KB)

Istotnym poleceniem jest również polecenie `sync` (bez argumentów), które wymusza zapis wszystkich danych z buforów dyskowych na dyski.

5. Swap

Jest to wolumin (rzadziej plik) na który system operacyjny zapisuje rzadko używane strony pamięci operacyjnej po to, aby zwolnić fizyczną pamięć operacyjną dla innych celów. Pozwala to na efektywniejsze użycie pamięci operacyjnej, ale niesie ze sobą groźbę znacznego spowolnienia pracy, gdy strony zapisane na dysku będą potrzebne (muszą być z powrotem załadowane do pamięci).

W systemie linux partycja przeznaczona na swap powinna mieć ustawiony odpowiedni typ. Przed użyciem wolumin swap trzeba sformatować za pomocą mkswap, np.: `mkswap /dev/hda1`

Przyłączenie swap dokonuje się za pomocą komendy swapon, np.: `swapon /dev/hda1`
Odłączenie swap dokonuje się za pomocą komendy swapoff, np.: `swapoff /dev/hda1`
przy czym strony pamięci z odłączanego woluminu są automatycznie przerzucane do RAM albo do innych woluminów (muszą się zmieścić).

Wykaz aktywnych woluminów swap i informację o ich zajętości można uzyskać komendą `swapon -s`

6. Loopback

Jest to pseudo-urządzenie o nazwie `/dev/loopN` (N=0, 1, ...), które pozwala na korzystanie ze zwykłego pliku o ustalonym rozmiarze tak, jak z urządzenia blokowego. Jest ono często stosowane do korzystania i tworzenia obrazów dyskietek/partycji dyskowych/CD/DVD bez ich zapisywania na odpowiedni nośnik. Urządzenie to umożliwia również utworzenie i korzystanie z zaszyfrowanych woluminów.

Przykład:

Dysponujemy obrazem płyty CD o nazwie `moja_plyta.iso` . Chcemy skorzystać z plików zawartych w tym obrazie bez nagrywania go na CD.

Aktywujemy pseudo-urządzenie `/dev/loop0` za pomocą komendy
```
losetup /dev/loop0 moja_plyta.iso
```

Od tego momentu urządzenie `/dev/loop0` zachowuje się jak CD-ROM z włożoną płytą CD.

Deaktywacja:
```
losetup -d /dev/loop0
```

Wyświetlenie informacji:
```
losetup -a
```

Systemy inne niż linux zwykle dysponują odpowiednikiem loopback o nieco zmienionej nazwie i nieco zmienionych komendach do jego obsługi.

7. Usuwanie danych

Zwykłe skasowanie plików nie usuwa danych z dysku. Podobnie sformatowanie partycji pozostawia większość danych na miejscu. Chcąc usunąć dane z dysku należy nadpisać istniejące dane innymi (np. zapisać partycję zerami). Tak skasowane dane stają się niedostępne dla przeciętnych osób, jednakże specjalistyczne laboratoria potrafią odzyskać wielokrotnie nadpisane dane na podstawie analizy pozostałości magnetycznych na dysku. Zapobiec temu może wielokrotne (nawet do 26 razy) nadpisanie danych za pomocą odpowiednio dobranych sekwencji bitowych, zastosowanych w losowej kolejności do różnych sektorów dysku. Istnieje specjalistyczne oprogramowanie do tego celu, jednym z przykładów może być pakiet PGP (www.pgp.com). Dokumentacja tego programu wspomina o fakcie, że komercyjne firmy zajmujące się odzyskiwaniem danych potrafiły odzyskać dane nadpisane aż 9 razy.

Usuwając dane warto pamiętać o tym, że resztki danych mogą się znajdować się nie tylko na głównym dysku/partycji z danymi ale także na dysku na którym znajduje się katalog plików tymczasowych (w linux'ie jest to katalog /tmp) oraz na partycji lub pliku stronicowania (swap).

8. Pytania kontrolne

1. Ile partycji podstawowych może być utworzonych (maksymalnie) na dysku twardym z tablicą partycji DOS
2. Na dysku /dev/hda są utworzone: partycja podstawowa /dev/hda1 oraz partycja rozszerzona /dev/hda2 zawierająca jeden dysk logiczny. Podaj jego oznaczenie.
3. Co umożliwia technologia SMART?
4. Jakie oznaczenie będzie miał dysk podłączony jako slave na drugim kanale ATA.
5. Jaka jest podstawowa różnica między adresacją CHS a LBA?
6. Co to jest MBR, jakie informacje zawiera ? (nie ucz się rozmiarów poszczególnych pól!)
7. Jakie informacje zawiera opis partycji? (nie ucz się rozmiarów poszczególnych pól!)
8. Wymień znane Ci wady i zalety systemu plików FAT.
9. Dlaczego jest istotne dobranie właściwej liczby węzłów w tworzonym systemie plików ext2?
10. Co to jest urządzenie loopback?
11. W jaki sposób można usunąć dane z dysku?
12. Jakie znasz systemy plików stosowane na dyskach CD-R oraz CD-RW
13. Do czego służy partycja swap?
14. Jakie informacje są zawarte w węźle pliku?
15. Gdzie jest przechowywana nazwa pliku (w systemie ext2)?
16. Co to jest /dev/hdb2?
17. Co to jest /dev/hda
18. Co to jest limit twardy i limit miękki (quota)?

Na zajęcia proszę przynieść płytę CD zawierającą dowolne dane.

9. Ćwiczenia

--- Ostrzeżenie !!! ---

Przedstawiony tekst zawiera komendy, które **mogą spowodować utratę danych** zapisanych na dyskach twardych fizycznie przyłączonych do komputera. **Dotyczy to również sytuacji, w której komendy są wydane w linuxie uruchomionym z CD!!!**

Poniższe ćwiczenia należy wykonywać na wirtualnej maszynie AKiSO lub SO. W systemie SO należy sprawdzić, czy maszyna wirtualna używa fizycznego urządzenia. Jeśli nie to zmienić ustawienia. Jeśli komputer PC nie ma napędu CD/DVD należy zamiast fizycznego urządzenia podmontować z poziomu systemu hosta maszyny wirtualnel plik z obrazem systemu CentOS (do ściągnięcia ze strony: http://ftp.agh.edu.pl/centos/7/isos/x86_64/).

Ćwiczenie 1:
W używanym systemie płyta CD jest montowana automatycznie po włożeniu do napędu w katalogu `/media/UDF Volume (w SO: /run/media/student/'nazwa voluminu')`. Można to sprawdzić poleceniem `mount`.

Odłącz (odmontuj) CD: `umount /media/UDF\ Volume (w SO: umount /run/media/student/'nazwa voluminu').`
Utwórz katalog `/tmp/cdrom` i podlacz w tym katalogu płytę CD.
`mount /dev/cdrom /tmp/cdrom (w SO: /dev/sr0 /tmp/cdrom)`
Zauważ, że od tej chwili nie można wyjąć płyty CD z napędu.

- Wyświetl podłączone woluminy komendą `mount` (bez parametrów). Zwróć uwagę na **typ systemu plików** na CD-ROM.
- Wyświetl miejsce zajęte/dostępne na woluminach komendą `df -k`.
- Wyświetl zawartość katalogu `/tmp/cdrom`
- Spróbuj skorzystać z plików zapisanych na CD
- Zmień katalog bieżący na `/tmp/cdrom`
- Odłącz `/tmp/cdrom` `poleceniem umount /tmp/cdrom` (nie uda się)
- Sprawdź, kto używa katalogu `/tmp/cdrom` poleceniem `lsof /tmp/cdrom`
- Zmień katalog bieżący na `/tmp`
- Odłącz `/tmp/cdrom` poleceniem `umount /tmp/cdrom` (powinno się udać)
- Wyświetl podłączone woluminy.
- Wyjmij płytę z napędu. Możesz nacisnąć przycisk na CD-ROM'ie albo wydać polecenie `eject /dev/cdrom`

Ćwiczenie 2:
Włóż dyskietkę do napędu (w maszynie wirtualnej AKiSO dyskietka jest już włożona, w maszynie SO należy ją uaktywnić przez polecenie 'Connect', które jest dostępne na naciśnięciu prawego klawisza myszy na ikonie dyskietki w prawym górnym ekranie maszyny wirtualnej).

- Poleceniem `file -s /dev/fd0` sprawdź **typ systemu plików** na dyskietce.
- Utwórz katalog `/tmp/floppy` i podłącz w nim dyskietkę poleceniem:
  `mount -t vfat /dev/fd0 /tmp/floppy`
- Wyświetl podłączone woluminy.
- Sprawdź, jaka jest zawartość dyskietki
- Na dyskietce znajduje się plik tekstowy `readme.txt` utworzony w systemie Windows, spróbuj go edytować za pomocą edytora `vi` z opcją `-u NONE` (`vi -u NONE nazwa_pliku`). Zauważ, że na końcu każdej linii jest dziwny znak (`^M`). Wynika to z różnych konwencji zapisu plików tekstowych w Windows (koniec linii to dwa znaki oznaczane w notacji języka C jako „`\r\n`") i w UNIX (koniec linii to tylko znak „`\n`") i powoduje trudne do zlokalizowania problemy. Przenosząc pliki tekstowe między systemami DOS/Windows i UNIX należy dokonać konwersji końców linii. Czasem potrafi to zrobić program, który kopiuje pliki, np. klient ftp albo ssh, czasem trzeba to zrobić ręcznie za pomocą poleceń `dos2unix` lub `unix2dos`.
- utwórz na dyskietce dowolny plik

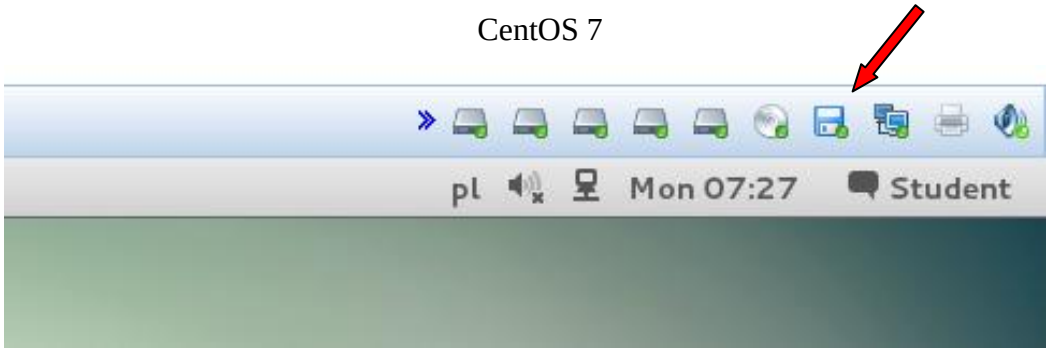
- zwróć uwagę na prawa dostępu na tym pliku, system FAT nie wspiera praw dostępu na plikach, więc dla zachowania zgodności z konwencją UNIX'a wszystkie pliki i katalogi mają ustawione pewne standardowe prawa dostępu i standardowego właściciela. Prawa te można ustawiać opcjami komendy `mount`
- odłącz `/tmp/floppy`
- podłącz ponownie dyskietkę używając polecenia:
  `mount -o uid=1000,gid=1000,umask=077 /dev/fd0 /tmp/floppy`
- sprawdź teraz właściciela plików i katalogów w `/tmp/floppy`
- zauważ (`mount` bez parametrów), że dyskietka cały czas jest podłączana w trybie read-write. Możesz ją podłączyć od razu w trybie read-only podając opcję `ro` (`rw` powoduje montowanie w trybie read-write), albo „w locie" zmienić to ustawienie komendą
  `mount -o ro,remount /tmp/floppy`

Ćwiczenie 3:

- zapisz na dyskietce (czyli w katalogu `/tmp/floppy`) plik wypełniony zerami o długości 300KB
- w razie wystąpienia błędów usuń ich przyczynę i spróbuj ponownie
- poczekaj aż zgaśnie lampka stacji dyskietek

CentOS 6

CentOS 7



- wykonaj polecenie `sync`
- zaobserwuj, że w tym momencie dane zostają faktycznie zapisane na dyskietkę (świeci lampka)
- zapisz na dyskietce kolejny plik o rozmiarze 300KB
- odłącz poleceniem `umount` dyskietkę, zaobserwuj, że odłączenie spowodowało zapis wszystkich pozostałych danych na dyskietce

Ćwiczenie 4:

Sprawdź, czy system plików na dyskietce jest poprawny za pomocą polecenia `fsck -v /dev/fd0` (opcja `-v` powoduje wyświetlanie dodatkowych informacji)

Ćwiczenie 5:

- utwórz nowy system plików FAT12 na dyskietce poleceniem `mkfs -t vfat -v /dev/fd0` (opcja `-v` powoduje wyświetlanie dodatkowych informacji)
- podłącz nowo stworzony system plików i wyświetl ilość dostępnego miejsca na dyskietce. Zauważ, że straciłeś ok. 16KB z oryginalnej pojemności (1440KB)
- odłącz system plików
- utwórz teraz na dyskietce system plików ext2 poleceniem `mkfs -t ext2 -v /dev/fd0`
  zwróć uwagę na tekst:
  `72 blocks (5.00%) reserved for the super user`

- podłącz nowo stworzony system plików i wyświetl ilość dostępnego miejsca na dyskietce komendą `df -k`. Należy zauważyć, że system podaje rozmiar woluminu 1412KB. Z oryginalnego rozmiaru 1440 KB tracimy 28KB na struktury systemu plików (między innymi tablice węzłów). Następnie, `df -k` podaje, że używane jest 13KB. Miejsce to zajmują pliki specjalne - katalogi: `.` oraz `lost+found` (co można sprawdzić poleceniem `ls -la`). Raportowane dostępne miejsce to 1327KB. Brakujące 72KB to miejsce zarezerwowane dla root'a.
- zwróć również uwagę, że w katalogu `/tmp/floppy` jest umieszczony systemowy katalog `lost+found`, specyficzny dla Unix'owych systemów plików. Jeżeli `fsck` w czasie naprawy systemu plików znajdzie bloki oznaczone jako zajęte, ale nie przypisane do żadnego z plików, to utworzy dla każdego z tych bloków plik w `lost+found`, co pozwala na ręczną analizę i wykorzystanie danych zapisanych w tych blokach.
- odłącz `/dev/fd0` i uruchom `fsck -v /dev/fd0` zauważ, że uruchomiła się wersja fsck dostosowana do systemu plików ext2

Ćwiczenie 6:
Utwórz system plików ext2 z maksymalną liczbą węzłów 16.
`mkfs -t ext2 -N 16 /dev/fd0`
Liczba wolnych węzłów (5) jest raportowana przez `dumpe2fs /dev/fd0` (11 wezłów jest zarezerwowane dla systemu). Zauważ, że polecenie `df -k` raportuje teraz rozmiar woluminu 1433 KB (na struktury systemu plików tracimy teraz nie 28KB, ale 7KB) co jest spowodowane mniejszym rozmiarem tablicy węzłów.

Podłącz wolumin w `/tmp/floppy`
Poleceniem `touch` spróbuj utworzyć 6 plików w `/tmp/floppy`.
Przy tworzeniu szóstego pliku system powinien zgłosić brak wolnego miejsca na woluminie (tak naprawdę brak wolego miejsca w tablicy węzłów). Sprawdź komendą `df -k`, że na woluminie `/dev/fd0` jest nadal dostępne miejsce. Możemy je zająć zapisując dane do istniejących plików. Tworzenie nowych plików jest niemożliwe.

Odłącz `/dev/fd0`.

BARDZO WAŻNE — ZAPAMIĘTAĆ !!!

NIE WOLNO zapisywać i odczytywać danych bezpośrednio do/z urządzeń (np. komendą dd of=/dev/fd0 ...) jeżeli konkretne urządzenie jest podłączone komendą mount albo w jakikolwiek inny sposób jest używane przez system (np. jest to aktywna partycja swap albo urządzenie stanowi aktualnie część macierzy RAID). Nieprzestrzeganie zakazu zapisu grozi uszkodzenien/utratą danych i zawieszeniem systemu. Nieprzestrzeganie zakazu odczytu powoduje, że możemy odczytać niespójny obraz dysku.
Ćwiczenie 7: Loopback
Plik `/root/test.iso` zawiera obraz jednej z płyt instalacyjnych systemu CentOS (pobrany z sieci):

- Poleceniem `losetup /dev/loop0 /root/test.iso` utwórz urządzenie loopback połączone z tym plikiem.
- Podłącz `/dev/loop0` w katalogu `/tmp/cdrom`, sprawdź jego zawartość.

- Poleceniem `dd bs=512 count=2880 if=/dev/zero of=/tmp/floppy.img` utwórz pusty obraz dyskietki.
- Utwórz urządzenie `/dev/loop1` połączone z plikiem `/tmp/floppy.img`

- Wyświetl urządzenia loopback poleceniem `losetup -a`
- Zlikwiduj `/dev/loop0` poleceniem `losetup -d /dev/loop0`

- Utwórz system plików FAT na `/dev/loop1`
- Podłącz `/dev/loop1` do katalog `/tmp/floppy`
- Nagraj jakiś plik na `/tmp/floppy`
- Odłącz `/dev/loop1`
- Zlikwiduj urządzenie `/dev/loop1` poleceniem `losetup -d /dev/loop1`
- Skopiuj za pomocą `dd` obraz `floppy.img` na dyskietkę.
- Podłącz dyskietkę w jakimś katalogu. Sprawdź, czy jest na niej plik, który ładowałeś do obrazu dyskietki.

Ćwiczenie 8: Partycjonowanie dysku
Uruchom `fdisk` poleceniem `fdisk /dev/sdb`
Za pomocą komendy **`p`** wyświetl informację o partycjach.

Utwórz `partycję /dev/sdb1` obejmującą całe dostępne miejsce (polecenie **n**)
Zakończ pracę fdisk'a poleceniem **w**

Sformatuj (polecenie `mkfs`) nowo utworzoną partycję w systemie jako ext3.

Poleceniem `dumpe2fs` wyświetl informacje o partycji:
```
dumpe2fs /dev/sdb1 | less
```

Zwróć uwagę na wartości parametrów „Maximum mount count" oraz „Next check after". Odpowiadają one za okresowe sprawdzanie struktury dysku przez fsck.

Poleceniem `tune2fs` przestaw maksymalną liczbę podłączeń, po której wystąpi automatyczne sprawdzanie dysku oraz maksymalny czas między sprawdzeniami na 0 (=wyłącz automatyczne sprawdzanie).
```
tune2fs -c 0 -i 0 /dev/sdb1
```

Automatyczne sprawdzanie dysków nie jest częścią procesu podłączania woluminu (mount). Prawie każdy system unix uruchamia przy starcie polecenie `fsck` dla każdego podłączanego woluminu. Jeżeli wolumin jest „czysty" (był poprawnie odłączony, był podłączany mniej niż ustawiona liczba razy i nie minął ustawiony okres czasu pomiędzy sprawdzaniami) to `fsck` kończy pracę natychmiast. W przeciwnym razie następuje czasochłonne sprawdzanie dysków. Ze względu na to, że taka operacja znacznie wydłuża czas przy starcie systemu wielu administratorów decyduje się na pominięcie okresowego sprawdzania woluminów ustawiając w/w parametry. Jest to szczególnie powszechne przy systemach z „dziennikiem".

UWAGA!!
Poniższe ćwiczenia należy wykonywać **z konta root'a na linuxie wystartowanym z sieci**.

Po wykonaniu tych ćwiczeń, około 10 min przed końcem zajęć należy zresetować komputer i ponownie zainstalować system Windows na dysku twardym. W tym celu należy na ekranie wyboru systemu operacyjnego wpisać labreinstall i zatwierdzić ładowanie obrazu systemu. Po załadowaniu obrazu należy uruchomić Windows celem dokończenia konfiguracji systemu.

Ćwiczenie 9: Mechanizm SMART
Za pomocą polecenia `smartctl -a /dev/sda | less` wyświetl wszystkie informacje SMART dotyczące urządzenia `/dev/sda`. (opcje `-c`, `-A`, `-l` wyświetlają poszczególne części tego zestawienia, opcja `-a` wyświetla całość informacji).

W szczególności sprawdź, jaka jest temperatura dysku, ile godzin dysk pracował, ile razy był włączany, ile sektorów było przeniesionych (reallocated sector count), czy jest jakiś sektor w trakcie realokacji (current_pending_sector).

Za pomocą komendy `smartctl -t short /dev/sda` zleć dyskowi wykonanie krótkiego autotestu. Autotest wykonuje się w tle (nie przeszkadza w normalnym korzystaniu z dysku), aktualny stan wykonania testu można uzyskać wykonując polecenie `smartctl -c /dev/sda`, jest on pokazany w części opisanej jako „Self-test execution status".

Za pomocą komendy `smartctl -o on /dev/sda` zleć dyskowi wykonywanie pełnego skanowania powierzchni dysku automatycznie co pewien okres czasu. Takie ustawienie nie zmienia nic z punktu widzenia użytkownika dysku (nie przeszkadza w pracy) a może spowodować wczesne wykrycie pogorszonej jakości zapisu w niektórych sektorach dysku (zanim ujawnią się błędy).

Ćwiczenie 10: Testowanie szybkości dysku (opcjonalne)
Podłącz partycję pod dowolnie wybrany katalog i zapisz na nią plik o rozmiarze 512MB (jest to ok. 2 razy więcej niż rozmiar pamięci RAM w komputerach w laboratorium, co w połączeniu z poleceniem `sync` zmniejsza wpływ cache dyskowego na wynik). Sprawdź czas potrzebny na zapisanie tego pliku przy zapisie z różną wielkością bloku (w poniższych komendach ważne są spacje!)

```
time { dd bs=512 if=/dev/zero of=plik count=1024x1024 ; sync ; }
time { dd bs=1k if=/dev/zero of=plik count=512x1024 ; sync ; }
time { dd bs=4k if=/dev/zero of=plik count=128x1024 ; sync ; }
time { dd bs=16k if=/dev/zero of=plik count=32x1024 ; sync ; }
time { dd bs=64k if=/dev/zero of=plik count=8x1024 ; sync ; }
```

Sprawdź także czas odczytu tak utworzonego pliku z różnymi wielkościami bloków. Pamiętaj, aby przed każdym odczytem odmontować i podmontować wolumin `/dev/sdb1`, co spowoduje wyczyszczenie cache dyskowego.

```
time dd bs=512 if=plik of=/dev/null
time dd bs=1k if=plik of=/dev/null
time dd bs=2k if=plik of=/dev/null
time dd bs=4k if=plik of=/dev/null
time dd bs=16k if=plik of=/dev/null
time dd bs=64k if=plik of=/dev/null
```

Czas oznaczony jako „real" to rzeczywisty czas wykonywania komendy. Czasy „user" i „sys" pokazują zużycie czasu procesora do wykonania (w ramach komendy) procedur użytkownika i systemu. Zauważ, że operacje przy użyciu małych bloków danych są nieefektywne gdyż dają niski transfer z dysku przy bardzo dużym obciążeniu procesora. Jest to spowodowane tym, że dla każdego przesyłanego bloku musi zostać wywołana odpowiednia systemowa procedura odczytu/zapisu z dysku. Pamiętaj o tym pisząc programy, które intensywnie korzystają z dysku.

Ćwiczenie 11 (opcjonalne):

Odłącz wolumin `/dev/sda1`. Skasuj partycję `/dev/sda1` i utwórz na dysku `/dev/sda` trzy partycje, każda o rozmiarze około 1GB umieszczone na początku, w środku i na końcu dysku (musisz sam policzyć od którego cylindra mają się zaczynać te partycje).

Sformatuj wszystkie trzy partycje. Spróbuj użyć innego systemu plików niż ext3, np. xfs albo reiserfs (podaj typ systemu po opcji `-t` w poleceniu `mkfs`).

Utwórz trzy katalogi i podłącz w nich utworzone partycje.

Przetestuj czas zapisu i odczytu pliku o rozmiarze ok. 512MB (użyj bloku 64k albo większego) na każdym z tych woluminów i porównaj czasy.

Odłącz woluminy i podłącz je ponownie, aby opróżnić cache dyskowy.

Zmierz teraz czas jednoczesnego odczytu plików z dwóch końców dysku:

```
time dd bs=64k if=plik1 of=/dev/null & time dd bs=64k if=plik2 of=/dev/null
&
```

W idealnej sytuacji czas odczytu powinien wzrosnąć dwukrotnie (dwa pliki), jednakże ze względu na to, że dysk musi przemieszczać głowice między początkiem a końcem dysku, czas potrzebny na jednoczesne odczytanie dwóch plików jest znacznie dłuższy. Zależy on od m.in. rozmieszczenia danych (aktualnie testowane jest najgorsze) oraz parametrów dysku takich jak średni i maksymalny czas wyszukiwania. Generalnie dyski ATA (tanie, przeznaczone do użytku domowego) charakteryzują się średnim czasem wyszukiwania rzędu 8 ms, dyski SCSI (drogie, stosowane w serwerach) charakteryzują się średnim czasem wyszukiwania rzędu 1-2 ms. Na szybkość mają także inne czynniki: sposób kolejkowania żądań w systemie operacyjnym i w dysku, wielkość pamięci podręcznych w dysku i w systemie operacyjnym, prędkość obrotowa talerzy dysku i inne.

Laboratorium 4

- **Macierze RAID, Narzędzia LVM, Sieciowe systemy plików**

--- Ostrzeżenie !!! ---

Przedstawiony tekst zawiera komendy, które **mogą spowodować utratę danych** zapisanych na dyskach twardych fizycznie przyłączonych do komputera. **Dotyczy to również sytuacji, w której komendy są wydane w systemie linux uruchomionym z CD!!!**

1. /proc

Informacje w tym rozdziale zostały zaczerpnięte z Red Hat Linux 9.0 Reference Guide:
http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/ref-guide

Wirtualny system plików `proc`, podłączany w katalogu `/proc` zawiera pliki które umożliwiają dostęp do wewnętrznych struktur jądra systemu operacyjnego. Są to pliki wirtualne, które nie są nigdzie zapisane na dysku. Dla większości z nich raportowany rozmiar wynosi 0, mimo to można z nich czytać a czasem pisać. Odczyt z tych plików jest odczytaniem wewnętrznych zmiennych jądra systemu operacyjnego, zapis do tych plików ustawia wartości zmiennych jądra. W ten sposób można odczytać np. informacje o zainstalowanym procesorze czy też zainstalowanych urządzeniach PCI, można ustawić pewne wartości, np. nazwę komputera, włączyć/wyłączyć przekazywanie pakietów pomiędzy interfejstami sieciowmi (ruting).

Część z plików w `/proc` ma format tekstowy, zrozumiały bezpośrednio dla użytkownika, część ma format binarny wymagający dla zrozumienia go odpowiednich narzędzi.

Przykład 6

Wyświetlenie informacji o zainstalowanym procesorze:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Pentium(R) 4 CPU 2.80GHz
stepping        : 9
cpu MHz         : 2798.724
cache size      : 512 KB
```

--- dalsza część wyników została pominięta

Przykład 2

Zmiana nazwy komputera (komenda hostname podaje aktualną nazwę komputera)

```
# hostname
Knoppix
# echo test.kt.agh.edu.pl > /proc/sys/kernel/hostname
# hostname
test.kt.agh.edu.pl
```

Zainteresowanych większą ilością informacji odsyłamy do dokumentacji.

2. RAID

Ideą RAID (Redundant Array of Inexpensive Disks) jest połączenie wielu dysków w jedną macierz dyskową, która jest szybsza, bardziej niezawodna i ma większą pojemność od pojedynczego dysku. Istnieją dwie podstawowe odmiany RAID - sprzętowa i programowa.

Odmiana sprzętowa (hardware RAID) wymaga zastosowania specjalnego kontrolera RAID do którego są podłączone dyski. Kontroler zajmuje się wówczas całkowicie obsługą RAID. Z punktu widzenia systemu operacyjnego macierz sprzętowa jest najczęściej postrzegana jako dysk SCSI (system nie odróżnia jej od

zwykłego dysku). Zaletą sprzętowego RAID jest to, że system operacyjny (i procesor) nie jest obciążony obsługą RAID oraz to, że korzysta się z macierzy RAID jak ze zwykłego dysku twardego. Nie ma więc problemów np. ze startem komputera z macierzy RAID. Wadą sprzętowego RAID jest koszt kontrolera.

W przypadku odmiany programowej RAID (software RAID) w systemie operacyjnym jest instalowane oprogramowanie, który tworzy macierz z podanych urządzeń blokowych (dysków), zajmuje się jej obsługą i prezentuje ją systemowi jako odrębne urządzenie blokowe. Główną zaletą programowego RAID jest brak konieczności zakupu specjalistycznego sprzętu, wadami - obciążenie procesora i magistral komputera obsługą RAID oraz uciążliwości w konfiguracji (np. macierz RAID jest dopiero dostępna po załadowaniu oprogramowania - nie działa w początkowej fazie ładowania systemu operacyjnego).

Istnieje 8 podstawowych poziomów RAID: Linear (zwany czasem concatenation), 0, 1, 2, 3, 4, 5, 6. Nie wszystkie z nich oferują jednocześnie szybkość, niezawodność i pojemność większą od pojedynczego dysku.

Poniżej przedstawiono krótką charakterystykę poziomów Linear, 0, 1, 4, 5 i 6. Poziomy 2 oraz 3 są bardzo rzadko używane, w związku z tym zostały pominięte w przedstawionym opisie.

Linear

Jest to „sklejenie" co najmniej 2 woluminów o dowolnych rozmiarach w jeden wolumin o większym rozmiarze. Pojemność macierzy jest sumą pojemności woluminów składowych, brak zysku na prędkości, awaria dowolnego woluminu składowego oznacza awarię macierzy.

Przykład: Przedstawiona macierz typu linear jest złożona z dwóch woluminów - odwzorowanie miejsca na macierzy na woluminy składowe pokazano na poniższym schemacie:

Linear	0	1

Wolumin 0	0
Wolumin 1	1

RAID 0 - Stripping

Wymaga co najmniej dwóch woluminów, zwykle muszą mieć one jednakowy rozmiar. Macierz składa się z bloków (stripe) o rozmiarze od kilku do kilkudziesięciu KB, rozmieszczonych na dwóch lub więcej woluminach składowych. Pojemność macierzy jest sumą pojemności woluminów składowych, szybki odczyt, szybki zapis, awaria dowolnego woluminu składowego oznacza awarię macierzy.

Przykład: Przedstawiona macierz typu RAID 0 jest złożona z dwóch woluminów - odwzorowanie miejsca na macierzy na woluminy składowe pokazano na poniższym schemacie. Kolejne bloki są umieszczane na przemian na woluminach składowych (parzyste na woluminie 0, nieparzyste na woluminie 1).

RAID 0	0	1	2	3	4	5	6	7	8	9

Wolumin 0	0	2	4	6	8
Wolumin 1	1	3	5	7	9

RAID 1 - Mirroring

Wymaga co najmniej dwóch woluminów. Każdy z woluminów składowych przechowuje identyczną kopię danych. Pojemność macierzy jest równa pojemności najmniejszego woluminu, odczyt jest zwykle szybki (rozkłada się na wszystkie woluminy składowe, chyba że sterownik celowo wykonuje odczyty tylko z jednego woluminu), wolny zapis (konieczność zapisania danych równolegle na wszystkich woluminach), macierz pracuje poprawnie jeżeli jest sprawny co najmniej jeden wolumin składowy.

Przykład: Przedstawiona macierz typu RAID 1 jest złożona z dwóch woluminów - odwzorowanie miejsca na macierzy na woluminy składowe pokazano na poniższym schemacie. Nie ma podziału na bloki, każdy z woluminów zawiera dokładną kopię danych.

RAID 1	0

Wolumin 0	0
Wolumin 1	0

Ze względu na to, że każdy z woluminów składowych zawiera identyczną kopię danych, możliwe jest bezpośrednie podłączenie woluminu składowego i korzystanie z niego (należy korzystać tylko w trybie read-only, tryb read-write spowoduje rozsynchronizowanie danych na macierzy). Własność ta jest często wykorzystywana przy starcie systemu operacyjnego (w przypadku korzystania z programowych macierzy) - wówczas system startuje z pierwszego dysku (w trybie read-only), następnie uruchamia macierze i podłącza system plików z macierzy.

RAID 4 - Stripping with parity

Wymaga co najmniej 3 dysków o zbliżonym rozmiarze. Jeżeli N jest liczbą dysków to N-1 dysków przechowuje informacje podzielone na paski o rozmiarze od kilku do kilkudziesięciu KB (tak jak RAID 0), natomiast ostatni dysk przechowuje parzystość (sumę XOR) danych zawartych na pozostałych dyskach składowych.

Przykład: Przedstawiona macierz typu RAID 4 jest złożona z trzech woluminów - odwzorowanie miejsca na macierzy na woluminy składowe pokazano na poniższym schemacie. Bloki 0 macierzy jest umieszczony w bloku 0 woluminu 0, blok 1 macierzy jest umieszczony w bloku 0 woluminu 1, suma kontrolna powstała przez wykonanie bitowej operacji XOR pomiędzy blokami 0 i 1 jest umieszczona w bloku 0 woluminu 2. W analogiczny sposób są rozmieszczone dalsze bloki macierzy.

RAID 4	0	1	2	3	4	5	6	7	8	9

Wolumin 0	0	2	4	6	8
Wolumin 1	1	3	5	7	9
Wolumin 2	0×1	2×3	4×5	6×7	8×9

W razie awarii jednego z dysków dane na nim zawarte można obliczyć na podstawie zawartości pozostalych dysków i sumy kontrolnej zapisanej na ostatnim dysku (mówimy wówczas, że macierz pracuje w trybie zdregradowanym). W razie zapisu np. bloku 3 do obliczenia nowej sumy kontrolnej 2×3 jest wymagany odczyt bloku 2 albo znajomość starej sumy 2×3 i starego bloku 3 (czasem tak jest szybciej).

Pojemność macierzy to (N-1) razy pojemność jednego woluminu, odczyt jest szybki (tak jak RAID 0), zapis jest wolny ze względu na konieczność obliczenia sumy kontrolnej z czym wiąże się często konieczność odczytania danych z pozostałych dysków. Macierz pracuje poprawnie w razie awarii jednego z woluminów. Awaria dwóch woluminów oznacza awarię macierzy.

Ważne: w przypadku pracy w trybie zdegradowanym, jeżeli zostanie odcięte zasilanie w trakcie zapisu (albo w przypadku macierzy programowej zawiesi się system) to macierz RAID 4 traci spójność (uszkodzeniu ulegają dane zapisane w pasku). Przykład: macierz z rysunku powyżej pracuje bez woluminu 0. Przy zapisie bloku 1 musi zostać jednocześnie zapisana nowa suma kontrolna 0×1. Jeżeli zostanie zapisany tylko blok 1 bez aktualizacji sumy 0×1 to odczyt bloku 0 (obliczany z bloku 1 i sumy 0×1) zwróci bliżej nieokreślone dane.

RAID 5 - Stripping with distributed parity

Zasada działania jest identyczna jak w RAID 4, ale dane o parzystości są rozrzucone po dyskach, co poprawia wydajność. Występują różne algorytmy dystrybuowania parzystości po woluminach składowych, na ilustracji pokazano schemat left-symmetric (domyślny w linux'ie, zwykle najszybszy).

Przykład: Przedstawiona macierz typu RAID 5 jest złożona z trzech woluminów - odwzorowanie miejsca na macierzy na woluminy składowe pokazano na poniższym schemacie. Bloki 0 macierzy jest umieszczony w bloku 0 woluminu 0, blok 1 macierzy jest umieszczony w bloku 0 woluminu 1, suma kontrolna powstała przez wykonanie bitowej operacji XOR pomiędzy blokami 0 i 1 jest umieszczona w bloku 0 woluminu 2. Bloki 2 i 3 macierzy są umieszczone w blokach 1 woluminów 2 i 0, suma kontrolna jest umieszczona na woluminie 2. itd.

Taki sposób rozłożenia danych powoduje, że żądanie odczytu każdych 3 kolejnych bloków (np. 0, 1, 2) będzie skierowanie równolegle na trzy dyski.

RAID 5	0	1	2	3	4	5	6	7	8	9

Wolumin 0	0	3	4×5	6	9
Wolumin 1	1	2×3	4	7	8×9
Wolumin 2	0×1	2	5	6×7	8

Pojemność macierzy to (N-1) razy pojemność jednego woluminu, odczyt jest szybki, zapis jest wolny ze względu na konieczność obliczenia sumy kontrolnej (obliczenia, konieczność odczytania danych z pozostałych dysków), macierz pracuje poprawnie w razie awarii jednego z woluminów. Awaria dwóch woluminów oznacza awarię macierzy.

Ważne: w przypadku pracy w trybie zdegradowanym, jeżeli zostanie odcięte zasilanie w trakcie zapisu (albo w przypadku macierzy programowej zawiesi się system) to macierz RAID 5 traci spójność (uszkodzeniu ulegają dane zapisane w pasku). Przykład: macierz z rysunku powyżej pracuje bez woluminu 0. Przy zapisie bloku 1 musi zostać jednocześnie zapisana nowa suma kontrolna 0×1. Jeżeli zostanie zapisany tylko blok 1 bez aktualizacji sumy 0×1 to odczyt bloku 0 (obliczany z bloku 1 i sumy 0×1) zwróci bliżej nieokreślone dane.

RAID 6 - Stripping with double distributed parity

Zasada działania jest podobna do RAID 5, z tym, że zamiast jednego dysku nadmiarowego stosuje się dwa a macierz jest odporna na jednoczesną awarie dwóch dysków. Do obliczania informacji nadmiarowej (w RAID 5 był to bit parzystości) stosuje się różnemetody, np. metody oparte na kodach Reed'a-Solomona.

--

Macierze RAID 4 i RAID 5 są odporne na pojedynczą awarię. W razie jej wystąpienia macierz pracuje dalej w tzw. trybie zdegradowanym. Wszystkie dane są normalnie dostępne, ale macierz pracuje nieco wolniej i nie jest odporna na kolejną awarię. Uszkodzony dysk należy wówczas jak najszybciej wymienić. Wymiana może się odbyć bez wyłączania zasilania jeżeli pozwala na to sprzęt (kontroler obsługuje tzw. hot swap), w pozostałych przypadkach konieczne jest wyłączenie i ponowne uruchomienie systemu. Aby skrócić czas w którym macierz jest zdegradowana możliwe jest zdefiniowanie dodatkowego woluminu jako tzw. hot spare. Taki wolumin nie jest używany w normalnej sytuacji, natomiast w razie awarii jakiegoś woluminu jest on automatycznie podłączany do macierzy zamiast uszkodzonego. Zwykle możliwe jest współdzielenie dysku hot spare przez kilka macierzy. Po wymianie macierz rozpoczyna rekonstrukcję danych na wymienionym dysku. Rekonstrukcja jest również wykonywana przy tworzeniu macierzy oraz w przypadku niepoprawnego zamknięcia systemu (awaria zasilania). W trakcie rekonstrukcji macierz pracuje normalnie, ale jest nieco wolniejsza i nieodporna na awarie.

Macierz RAID 6 jest odporna na pojedynczą i podwójną awarię. W razie ich wystąpienia macierz pracuje w trybie zdegradowanym, zwykle oznacza to wolniejszy odczyt i dużo wolniejszy zapis. Podobnie jak dla macierzy RAID 4 i RAID 5 można zdefiniować dodatkowy wolumin jako hot-spare.

Macierz RAID 1 złożona z N dysków jest odporna na N-1 równoczesnych awarii. W razie ich wystąpienia macierz pracuje w trybie zdegradowanym, zwykle oznacza to wolniejszy odczyt i szybszy zapis. Podobnie jak dla poprzednich typów macierzy można zdefiniować dodatkowy wolumin jako hot-spare.

Macierze RAID 1, 4, 5 powinny mieć woluminy umieszczone na różnych dyskach/urządzeniach fizycznych. Jeżeli dwa woluminy z jednej macierzy są umieszczone na tym samym dysku, to w razie awarii tego dysku macierz przstanie działać (2 woluminy są uszkodzone). Ponadto umieszczenie dwóch woluminów składowych tych macierzy na 1 dysku fizycznym spowoduje znacznie wolniejszy dostęp do danych (ważne również dla macierzy RAID 0). Jedyną macierzą przy której ma sens umieszczenie dwóch woluminów składowych na 1 dysku fizycznym jest macierz typu Linear.

Z macierzy sprzętowych można korzystać w każdym systemie. Dostępność macierzy programowych jest różna, np. Linux i Solaris obsługują macierze programowe RAID, system Windows XP ma bardzo ubogie możliwości w tym zakresie, większe możliwości ma Windows Server.

W systemie Linux można korzystać z programowych macierzy RAID. Macierze są dostępne jako urządzenia blokowe /dev/mdX gdzie X=0,1,... (skrót md pochodzi od „meta device")

Partycje dyskowe które będą używane jako części składowe macierzy powinny mieć typ 0xfd, dzięku któremu będą one rozpoznane i podłączone do właściwych macierzy w czasie startu systemu (jeżeli chcemy tego uniknąć wystarczy użyć innego typu partycji, np. 0x83 - Linux).

Na końcu każdego woluminu używanego przez Linux Software Raid jest rezerwowany obszar, w którym są zapisane informacje o macierzy do której to urządzenie należy. Przy starcie systemu jest więc możliwe automatyczne „poskładanie" wszystkich macierzy z właściwych części.

Do tworzenia i obsługi macierzy służy polecenie mdadm. Aktualny stan macierzy można obejrzeć wyświetlając zawartość pliku /proc/mdstat

W systemie Linux możliwa jest regulacja prędkości synchronizacji macierzy, dokonuje się tego na bieżąco odpowiednim wpisem do plików:
/proc/sys/dev/raid/speed_limit_min
/proc/sys/dev/raid/speed_limit_max

Bardziej szczegółowe informacje można uzyskać z: Jakob Østergaard, Emilio Bueso, „The Software-RAID HOWTO," http://www.tldp.org/HOWTO/Software-RAID-HOWTO.html

3. LVM

LVM - Logical Volume Manager jest pakietem oprogramowania służącym do bardziej elastycznego zarządzania przestrzenią dyskową w porównaniu z tradycyjnym podziałem na partycje. W sytuacji, w której często trzeba definiować woluminy o różnych rozmiarach, poszerzać istniejące woluminy jest to bardzo przydatne narzędzie, gdyż te zmiany mogą być dokonywane „w locie" bez konieczności restartowania systemu. Kosztem zastosowania LVM jest nieco wolniejsza praca dysków.

Terminologia:

- PV (physical volume) jest to wolumin na którym będą pisane dane. Może to być dowolne urządzenie blokowe, np. partycja dyskowa albo macierz RAID. PV jest dzielona na bloki o określonym rozmiarze (domyślnie 4MB) zwane PE (Physical Extents).

- VG (volume group) jest grupą złożoną z jednego lub więcej PV. Rozmiar VG jest równy sumie rozmiarów PV z których się składa dana VG. Wszystkie PV należące do danej VG mają taki sam rozmiar bloku PE.

- LV (logical volume) wydzielana z określonej VG. LV składa się z bloków LE (logical extents) o rozmiarze takim jak rozmiar bloku PE obowiązujący w danej grupie. Każda LV jest osobnym urządzeniem blokowym, które można sformatować za pomocą mkfs i podłączyć do drzewa katalogów.

Krótko mówiąc: łączymy urządzenia blokowe (PV) w jedną większą całość (VG) żeby potem móc ją swobodnie dzielić na LV.

Dodatkową zaletą LVM jest to, że można odłączyć daną PV „w locie". Przed odłączeniem należy zwolnić miejsce na odłączanej PV. Konieczne jest więc przeniesienie używanych PE na inne PV w ramach VG. Na pozostałych PV w ramach danej VG musi być wystarczająco dużo wolnego miejsca, żeby przyjąć dane z odłączanej PV.

Korzystanie z systemu LVM należy rozpocząć od przygotowania PV. Partycje przeznaczone na PV powinny mieć typ 0x8e, który pozwala na ich automatyczne rozpoznanie i podłaczenie do odpowiednich VG w czasie startu systemu. Każda PV musi być zainicjalizowana komendą `pvcreate`. Następnie należy utworzyć nową VG za pomocą komendy `vgcreate`. Od tego momentu można tworzyć i usuwać LV w ramach utworzonej VG oraz podłączać i odłączać PV do danej VG.

Bardziej szczegółowe informacje można uzyskać z: A.J. Lewis, „LVM HOWTO,"
http://www.tldp.org/HOWTO/LVM-HOWTO

4. NFS (mount, export)

Sieciowy system plików NFS został stworzony przez firmę Sun. Maszyna będąca serwerem NFS udostępnia innym maszynom, identyfikowanym przez adresy IP, katalogi. NFS zakłada, że numeryczne identyfikatory użytkowników na obu maszynach (serwer, klient) są zgodne, tzn. użytkownik o uid=100 po stronie klienta będzie miał dostęp jako użytkownik o uid=100 na serwerze. Utrzymanie zsynchronizaowanych uid na obu maszynach może być wykonywane „ręcznie," można się też posiłkować odpowiednimi usługami sieciowymi (np. NIS - Network Information Service). Wyjątkiem od powyższej zasady jest root (uid=0), który ze względów bezpieczeństwa jest traktowany przez serwer NFS jako uid=65534 (nobody) (tzn. tworzy pliki jako nobody, czyta pliki jako nobody).

NFS do transportu danych (plików) może korzystać z protokołów UDP (domyślnie) oraz TCP. Korzystanie z UDP ma tę zaletę, że w razie awarii serwera klienci mogą dość długo czekać (ok. tydzień) na ponowne uruchomienie serwera NFS, po czym podejmują bez problemu przerwane działania. Wadą NFS po UDP jest to, że korzysta on bardzo intensywnie z fragmentacji pakietów IP, co jest generalnie źle tolerowane przez zapory ogniowe (firewall). Korzystanie z NFS po TCP jest dobrze tolerowane przez zapory ogniowe i lepiej niż NFS po UDP radzi sobie z sieciami, w których łącza mają różne przepływności, jednakże niesie ze sobą inne wady. Awaria serwera powoduje zerwanie sesji NFS i konieczność odłączenia i ponownego podłączenia udostępnianego katalogu po stronie klienta. Wiąże się to z przerwaniem pracy aplikacji, które miały otwarte pliki na katalogu sieciowym. Ponadto NFS po TCP jest nieco wolniejszy od NFS po UDP przy pracy w szybkiej sieci lokalnej.

NFS składa się z kilku usług należących do rodziny RPC (Remote Procedure Call), które są rejestrowane w systemie przez demona `portmapper`.

Listę zarejestrowanych usług RPC można obejrzeć poleceniem `rpcinfo -p`. Ze względu na to, że mogą być równolegle uruchomione serwery NFS w kliku wersjach (wersja 2, 3, 4) usługi RPC mogą pojawiać się wielokrotnie.

Uruchomienie portmappera wystarcza do tego, by być klientem NFS. Katalogi sieciowe podłączamy poleceniem:

mount -t NFS -o opcje adres.serwera.nfs:/katalog_udostepniany lokalny_katalog

Opcje „mount"dla NFS można znaleźć w dokumentacji linux'a za pomocą `man mount` oraz `man 5 nfs`.

Konfiguracja serwera jest bardziej złożona. Podstawowe demony jakie muszą działać na serwerze to: portmapper oraz nfsd i mountd. Dodatkowo powinny działać lockd, statd i rquotad (można to sprawdzić komendą rpcinfo -p).

nfsd - zarządza procesami serwera NFS
mountd - obsługuje żądania podłączenia katalogów wydawane przez klientów nfs
lockd - obsluguje blokady plików (zapewnia wyłączność dostępu do danego pliku dla konkretnego klienta, jeśli jest to wymagane przez klienta) - w aktualnych wersjach jądra Linux'a proces lockd jest uruchamiany automatycznie przez jądro razem z nfsd
statd - nazwa jest nieco myląca, ten demon umożliwia klientom zauważenie faktu, że serwer wykonał nieoczekiwany restart (np. awaria zasilania)
rquotad - pozwala na informowanie klienta o aktualnych limitach miejsca na serwerze

Głównym plikiem konfiguracyjnym serwera NFS jest /etc/exports. Każda linia w tym pliku składa się z określenia katalogu, który jest udostępniany oraz informacji komu i na jakich zasadach jest udostępniany. Przy modyfikowaniu pliku /etc/exports należy bardzo uważać na wprowadzane spacje gdyż mogą one nieoczekiwanie zmienić faktyczne prawa klienta.

Przykład 2

Przykładowy plik /etc/exports

```
# cat /etc/exports
/tmp/a        149.156.203.0/24(ro,all squash,async)
/tmp/b        149.156.203.0/24 (rw,all_squash,async,insecure)
```

Nadmiarowa spacja w drugiej linii powoduje, że katalog /tmp/b jest udostępniany komputerom z sieci 149.156.203.0/24 z domyślnymi opcjami i **WSZYSTKIM komputerom z całego Internetu z opcjami (rw, all_squash,async,insecure) czyli DO ZAPISU!!!**

Aktualnie udostępniane katalogi można obejrzeć poleceniem exportfs -v

Warto pamiętać, że wszystkie usługi NFS korzystają z plików /etc/hosts.allow oraz /etc/hosts.deny, które regulują dostęp z sieci do usług uruchomionych na serwerze.

Przykład 3

Przykładowy plik /etc/hosts.allow, który dopuszcza do korzystania z usług NFS tylko komputery znajdujące się w sieci 149.156.203.0/24 oraz (WAŻNE) sam serwer NFS (127.0.0.1 = localhost)

```
# cat /etc/hosts.allow
ALL:    127.0.0.1
portmap: 149.156.203.0/255.255.255.0
lockd: 149.156.203.0/255.255.255.0
rquotad: 149.156.203.0/255.255.255.0
mountd: 149.156.203.0/255.255.255.0
statd: 149.156.203.0/255.255.255.0
```

Bardziej szczegółowe informacje można uzyskać z: T. Barr, N. Langfeldt, S. Vidal, T. McNeal, „Linux NFS-HOWTO,” http://www.tldp.org/HOWTO/NFS-HOWTO

5. SMB (mount, export)

SMB (server Message Block) jest protokołem stosowanym do udostępniania plików i drukarek w systemach Windows. Funkcjonalność SMB w systemach Linux zapewnia pakiet Samba, o bardzo szerokich możliwościach. W poniższym tekście zostanie przedstawiona najbardziej podstawowa funkcjonalność Samby, bardziej szczegółowe informacje można uzyskać ze strony www.samba.org (jest tam również dostępne HOWTO).

Korzystanie z udziałów sieciowych (network share) jest możliwe na dwa sposoby:

- za pomocą programu smbclient, który przypomina nieco klienta FTP, np.:
  smbclient ‘//labserv.kt.agh.edu.pl/public’
- podłączając udostępnione katalogi do drzewa katalogów (tylko w systemach Linux)
  mount -t cifs ‘//labserv.kt.agh.edu.pl/public’ /tmp/smb

W zależności od sytuacji (Windows server/Linux client, Linux serwer/Windows client, Linux serwer/Linux client) istnieją różne możliwości odwzorowywania użytkowników z jednego systemu w drugim systemie i przypisywania praw dostępu. Opis ten wykracza poza ramy niniejszego opracowania.

Serwerem samby jest para demonów nmbd/sbmd. Plik konfiguracyjny to zwykle /etc/samba/smb.conf

6. Przykład

Poniższy przykład jest nieco absurdalny, ale znakomicie pokazuje elastyczność rozwiązań spotykanych w Unix’ach. Przed zastosowaniem tego typu rozwiązań w rzeczywistej sytuacji należy je dobrze przetestować w różnych wariantach awarii.

Nasz serwer o nazwie A ma służyć jako serwer sieciowy do przechowywania krytycznie ważnych danych. Do dyspozycji mamy serwery B i C (umieszczone w osobnych budynkach) połączone szybką siecią z A. Dane powinny być na bieżąco zapisywane na wszystkich trzech serwerach. Prędkość dostępu do danych nie jest zbyt istotna.

Można to osiągnąć w następujący sposób:

Każdy z trzech serwerów ma katalogi /tmp/A, /tmp/B, /tmp/C utworzone na lokalnej macierzy RAID.

B i C udostępniają przez NFS katalogi /tmp/B oraz /tmp/C (odpowiednio).
Serwer A montuje te katalogi w swoich katalogach /tmp/B i /tmp/C, ponadto ma katalog /tmp/A założony na własnym dysku twardym.

Resztę operacji wykonujemy na serwerze A:

- tworzymy w każdym z katalogów /tmp/A, /tmp/B, /tmp/C plik o nazwie image (o rozmiarze odpowiadającym naszym potrzebom np. 100MB)
- za pomocą losetup wiążemy utworzone pliki z /dev/loop0 /dev/loop1 /dev/loop2
- tworzymy macierz RAID 1 (/dev/md0) z urządzeń /dev/loop0 /dev/loop1 /dev/loop2
- tworzymy PV na /dev/md0 i podłączamy do VG
- tworzymy dyski LV na VG, formatujemy je i udostępniamy użytkownikom za pomocą samby. Dane zapisane na każdym z tych LV są automatycznie pisane na trzech serwerach znajdujących się w trzech miejscach, przy czym każdy serwer pisze dane na dwóch dyskach jednocześnie.
- w razie awarii A dowolny serwer B lub C może przejąć jego rolę.

7. Pytania kontrolne

1. Co zawierają pliki w katalogu /proc
2. Omów zasadę działania RAID Linear
3. Omów zasadę działania RAID 0
4. Omów zasadę działania RAID 1
5. Omów zasadę działania RAID 4
6. Omów zasadę działania RAID 5
7. Do czego służy LVM?
8. Z czego można utworzyć PV (Physical Volume w LVM)
9. Co to jest LV? Jak LV jest widziana w systemie operacyjnym?
10. Wymień usługi składowe NFS.
11. Jakie są zalety i wady korzystania z protokołu UDP przy NFS
12. Jakie są zalety i wady korzystania z protokołu TCP przy NFS
13. W jaki sposób można ograniczyć dostęp z sieci do katalogów udostępnionych przez NFS?
14. W jaki sposób można na Linuksie skorzystać z plików udostępnionych przez komputer z Windows?

8. Ćwiczenia

--- Ostrzeżenie !!! ---

Przedstawiony tekst zawiera komendy, które **mogą spowodować utratę danych** zapisanych na dyskach twardych fizycznie przyłączonych do komputera. **Dotyczy to również sytuacji, w której komendy są wydane w linux'ie uruchomionym z CD!!!**

Poniższe ćwiczenia należy wykonywać na maszynie wirtualnej.

Zapamiętaj:

Jeżeli podłączyłeś system plików np. z /dev/sdb1 (mount /dev/sdb1 /jakis_katalog) to NIE WOLNO CI pisać bezpośrednio na /dev/sdb1 (w szczególności tworzyć tam nowego systemu plików za pomocą mkfs). Jeżeli chcesz utworzyć tam plik to tworzysz go w katalogu /jakis_katalog

Jeżeli podłączyłeś np. /dev/sdb1 do macierzy to NIE WOLNO CI korzystać bezpośrednio z tego urządzenia (/dev/sdb1). Używasz macierzy (np. /dev/md0)

Jeżeli podłączyłeś np. /dev/sdb1 do VG to NIE WOLNO CI korzystać bezpośrednio z tego urządzenia (/dev/sdb1). Używasz LV utworzonych na VG (np. /dev/lab/test)

Ćwiczenie 1:
Wyświetl informacje o zainstalowanym procesorze za pomocą `cat /proc/cpuinfo`
Wyświetl informacje o konfiguracji przerwań w systemie za pomocą `cat /proc/interrupts`
Wyświetl informacje o podłączonych urządzeniach SCSI za pomocą `cat /proc/scsi/scsi`

Ćwiczenie 2 – przygotowanie systemu:
Utwórz partycje `/dev/sdb1`, `/dev/sdc1`, `/dev/sdd1`, `/dev/sde1` o rozmiarze 200MB i ustaw typ partycji Linux RAID (`0xfd`).

Ćwiczenie 3:
Komendą `mdadm -C -l linear -n3 /dev/md0 /dev/sdb1 /dev/sdc1 /dev/sdd1` utwórz macierz `/dev/md0` typu linear składającą się z trzech partycji. Od tej chwili **nie wolno** używać bezpośrednio `/dev/sdb1 /dev/sdc1 /dev/sdd1` gdyż są one używane i zapisywane przez sterownik macierzy.

Partycje składowe miały po 200MB. Oblicz (szacunkowo) ile powinno być dostępnego miejsca na macierzy.

Wyświetl informacje o aktywnych macierzach za pomocą `cat /proc/mdstat`.
XXX blocks oznacza, że macierz ma rozmiar XXX KB. Czy zgadza się to z Twoimi obliczeniami?

Utwórz system plików ext3 na `/dev/md0`
`mkfs -t ext3 /dev/md0`

Utwórz katalog `/tmp/raid` i podłącz tam macierz komendą `mount /dev/md0 /tmp/raid`
Sprawdź ile jest wolnego miejsca na `/tmp/raid` (`df -k`)
Skopiuj do macierzy (czyli do katalogu `/tmp/raid`) jakiś plik.
Odłącz `/dev/md0` (`umount /dev/md0`)
Za pomocą `mdadm -S /dev/md0` zatrzymaj macierz.

Wyświetl informacje o aktywnych macierzach za pomocą `cat /proc/mdstat` Zauważ, że nie ma już informacji o `/dev/md0`.

Zatrzymaną macierz można ponownie uruchomić. Operacje zatrzymywania i uruchamiania macierzy nie powodują utraty danych zapisanych na macierzy. Uruchomienie macierzy polega na złożeniu części wcześniej istniejącej macierzy w działającą macierz. Wszystkie części, które zamierzamy złożyć muszą należeć do jednej macierzy, natomiast kolejność w jakiej są podawane jest dowolna. Macierz nie zostanie uruchomiona jeśli podane części składowe nie należą do jednej macierzy - jest to zabezpieczenie przed omyłkowym złożeniem

macierzy z przypadkowych części co spowodowałoby utratę danych. Macierz jest zwykle uruchamiana samoczynnie przy starcie systemu, ale można to zrobić za pomocą odpowiedniej komendy.

Należy zwrócić uwagę na istotną różnicę – URUCHOMIENIE macierzy zachowuje dane na niej zapisane. UTWORZENIE macierzy powoduje (w większości przypadków) utratę danych zapisanych wcześniej na urządzeniach, z których tworzymy macierz.

Uruchom ponownie macierz składając ją z istniejących części
`mdadm -A /dev/md0 /dev/sdb1 /dev/sdc1 /dev/sdd1`

Podłącz `/dev/md0` do `/tmp/raid`, sprawdź, czy plik który tam kopiowałeś da się przeczytać.

Odłącz od drzewa katalogów i zatrzymaj macierz `/dev/md0`.

Ćwiczenie 4:
Za pomocą komendy
`mdadm -C -l raid0 -n3 /dev/md0 /dev/sdb1 /dev/sdc1 /dev/sdd1`
utwórz macierz RAID 0.

Ponieważ `/dev/sdb1, /dev/sdc1 i /dev/sdd1` były częściami składowymi macierzy tworzonej w poprzednim ćwiczeniu, więc polecenie `mdadm` zapyta, czy na pewno chcesz utworzyć z nich nową macierz (jest to zabezpieczenie przed przypadkową pomyłką). Potwierdź tworzenie nowej macierzy.

Partycje składowe miały po 200MB. Oblicz (szacunkowo) ile powinno być dostępnego miejsca na macierzy.

Wyświetl informację o aktywnych macierzach (`cat /proc/mdstat`). Odczytaj jaki jest domyślny rozmiar bloku (`chunksize`) tej macierzy oraz jej rozmiar (XXX blocks - podane w 1KB blokach).

Utwórz system plików XFS na `/dev/md0` i podłącz macierz (`mount`) w katalogu `/tmp/raid`.
(przy tworzeniu systemu plików XFS możesz zostac poproszony o podanie opcji -f. Istotna jest tu kolejnośc opcji: najpierw -t xfs a potem -f)
Sprawdź ile jest wolnego miejsca na `/tmp/raid` (`df -k`)
Odłącz `/dev/md0` (`umount`)
Zatrzymaj macierz `/dev/md0` za pomocą `mdadm -S /dev/md0`

Ćwiczenie 5:
Za pomocą komend:
`cat /proc/sys/dev/raid/speed_limit_min`
`cat /proc/sys/dev/raid/speed_limit_max`

wyświetl jaka jest dopuszczalna minimalna i maksymalna prędkość rekonstrukcji macierzy.
Za pomocą komendy
`echo 2000 > /proc/sys/dev/raid/speed_limit_max`
ustaw maksymalną szybkość resynchronizacji macierzy na 2000KB/s.

Za pomocą komendy
`mdadm -C -l raid1 -n3 /dev/md0 /dev/sdb1 /dev/sdc1 /dev/sdd1`
utwórz macierz RAID 1.

Partycje składowe miały po 200MB. Oblicz (szacunkowo) ile powinno być dostępnego miejsca na macierzy.

Wyświetl informację o aktywnych macierzach (`cat /proc/mdstat`). Zauważ, że resynchronizacja macierzy przebiega z szybkością zbliżoną do maksymalnej dopuszczalnej. Czy rozmiar macierzy zgadza się z Twoimi obliczeniami?

Nie czekając na dokończenie resynchronizacji wykonuj dalsze polecenia:
(na maszynie wirtualnej resynchronizacja trwa bardzo krótko – prawdopodobnie nie uda Ci się zakończyć ćwiczenia przed zakończeniem resynchronizacji)
Utwórz system plików ext3 na `/dev/md0`.
Podłącz macierz `/dev/md0` w katalogu `/tmp/raid`.
Sprawdź ile jest wolnego miejsca na `/tmp/raid` (`df -k`)
Skopiuj do macierzy (czyli do katalogu `/tmp/raid`) jakiś plik.
Zauważ, że mimo trwającej rekonstrukcji możesz swobodnie pracować na macierzy.

Poczekaj aż macierz dokończy synchronizację. Zwróć uwagę na symbol [UUU] oznaczający, że macierz ma wszystkie trzy woluminy składowe sprawne.

W macierzach z nadmiarowością (RAID 1,4,5) można bez wyłączania macierzy odpiąć jeden z woluminów składowych i podpiąć w jego miejsce inny.

Wymuś na macierzy uznanie dysku `/dev/sdc1` jako uszkodzonego
```
mdadm -f /dev/md0 /dev/sdc1
```

Obejrzyj stan macierzy (`cat /proc/mdstat`) - macierz pracuje w trybie zdegradowanym, o czym świadczy znak podkreślenia zamiast litery U ([U_U])

Usuń wolumin `/dev/sdc1` z macierzy:
```
mdadm -r /dev/md0 /dev/sdc1
```

Obejrzyj stan macierzy (`cat /proc/mdstat`) - macierz pracuje nadal w trybie zdegradowanym.

Zwróć uwagę, że pliki zapisane na macierzy nadal są dostępne.

Wstaw do macierzy wolumin `/dev/sde1`
```
mdadm -a /dev/md0 /dev/sde1
```
Obejrzyj stan macierzy - rozpoczęła się resynchronizacja.
Poczekaj na zakończenie resynchronizacji.

Dodaj teraz wolumin `/dev/sdc1` do macierzy:
```
mdadm -a /dev/md0 /dev/sdc1
```
Obejrzyj stan macierzy. Zauważ, że system nie wykonuje resynchronizacji na `/dev/sdc1`. Ten wolumin jest traktowany jako „hot spare”.

Wymuś na macierzy uznanie woluminu `/dev/sdb1` jako uszkodzonego i usuń go z macierzy
```
mdadm -f /dev/md0 /dev/sdb1
mdadm -r /dev/md0 /dev/sdb1
```

Obejrzyj stan macierzy - rozpoczęła się automatycznie resynchronizacja na wolumin `/dev/sdc1`, po chwili macierz będzie pracować w trybie normalnym.

Zgodnie z zasadą działania macierzy RAID 1 każdy z dysków z tej macierzy przechowuje taką samą informację jak cała macierz. Wolumin `/dev/sdb1` został uznany za uszkodzony i odłączony od pracującej macierzy. Powinien zatem posiadać kopię danych z chwili, w której został odłączony (tak jakby komputer został zresetowany). Dodatkowo, system plików używany na macierzy to ext3, więc nie powinno być problemu ze spójnością zapisanych tam danych.
Utwórz katalog `/tmp/raidtest`. Podłącz wolumin `/dev/sdb1` do `/tmp/raidtest`. Obejrzyj pliki w katalogu `/tmp/raidtest`.

Właśnie w taki sposób można było (przed pojawieniem się LVM) wykonać spójną (wszystkie pliki zostały zamrożone w jednecj chwili) kopię zapasową plików w systemie, którego nie można wyłączać. Potrzebne do tego są trzy dyski, żeby w chwili wykonywania kopii zapasowej nie tracić odporności na awarie. Należy się oczywiście liczyć z faktem, że wolumin nie jest w 100% sprawny (nie był odmontowany w chwili odpięcia od macierzy) - zatem część danych może być uszkodzona. Obecnie taką funkcjonalność zapewnia pod linuksem LVM (snapshot volumes) a pod Windows mechanizm „Shadow Copies". Drugim zastosowaniem RAID 1 jest wykonanie „bezpiecznej" aktualizacji systemu - w przypadku gdy na RAID 1 znajduje się system operacyjny. Odłączając jeden z dysków mamy tam kopię zapasową systemu (działającą). Następnie aktualizujemy system. Jeżeli po aktualizacji system działa nadal - podłączamy z powrotem dysk do macierzy. Jeżeli aktualizacja się nie powiedzie uruchamiamy system z kopii na odłączonym dysku.

Odłącz `/dev/md0` oraz `/dev/sdb1` (`umount`)
Zatrzymaj macierz `/dev/md0` za pomocą `mdadm -S /dev/md0`

Ćwiczenie 6:
Za pomocą komendy
```
mdadm -C -l raid5 -n3 /dev/md0 /dev/sdb1 /dev/sdc1 /dev/sdd1
```
utwórz macierz RAID 5.

Partycje składowe miały po 200MB. Oblicz (szacunkowo) ile powinno być dostępnego miejsca na macierzy.
Sprawdź za pomocą `cat /proc/mdstat` jaka jest wielkość bloku (`chunksize`) oraz rozmiar macierzy. Czy zgadza się on z Twoimi obliczeniami?

Utwórz system plików ext2 na `/dev/md0`.
Podłącz macierz `/dev/md0` w katalogu `/tmp/raid`.
Sprawdź ile jest wolnego miejsca na `/tmp/raid` (`df -k`)
Skopiuj do macierzy (czyli do katalogu `/tmp/raid`) jakiś plik.
Zauważ, że mimo trwającej rekonstrukcji możesz swobodnie pracować na macierzy.

Poczekaj aż macierz dokończy synchronizację. Zwróć uwagę na symbol [UUU] oznaczający, że macierz ma wszystkie trzy woluminy składowe sprawne.

Odłącz `/dev/md0` (`umount`)
Zatrzymaj macierz `/dev/md0` za pomocą `mdadm -S /dev/md0`

Ćwiczenie 7:
Za pomocą fdisk'a zmień typ partycji `/dev/sdb1` - `/dev/sde1` na `0x8e` (Linux LVM)

Uwaga: typ partycji powinien być ustawiony zgodnie z tym co znajduje się bezpośrednio na partycji. Jeśli partycja będzie wykorzystana bezpośrednio do LVM to właściwym typem jest `0x8e` (Linux LVM). Jeśli natomiast partycja jest częścią macierzy RAID, a dopiero macierz jest używana jako PV w LVM to właściwym typem partycji jest `0xfd` (Linux RAID).

Utwórz PV na każdej z partycji `/dev/sdb1` - `/dev/sde1`
```
pvcreate /dev/sdb1 /dev/sdc1 /dev/sdd1 /dev/sde1
```

Wyświetl informację o PV `/dev/sdb1`:
```
pvdisplay /dev/sdb1
```

Utwórz VG o nazwie lab składającą się z PV `/dev/sdb1`
```
vgcreate lab /dev/sdb1
```

Wyświetl informację o dostępnych VG:
```
vgdisplay
```

Możesz także wykonać `vgscan` - to wyszuka wszystkie dostępne VG.

Na VG `lab` utworz LV o nazwie `test` i rozmiarze 150MB:
`lvcreate -L 150M -n test lab`
Zwróć uwagę na nazwę urządzenia. Jest to `/dev/nazwa_VG/nazwa_LV`.

Wyświetl informację o LV:
`lvdisplay /dev/lab/test`

Wyszukaj wszystkie LV:
`lvscan`

Załóż system plików `xfs` na utworzonej LV
Utwórz katalog `/tmp/test` i podłącz w nim stworzoną LV.
Wyświetl za pomocą `df -k` dostępne miejsce na stworzonej LV.

Podłącz `/dev/sdc1` i `/dev/sdd1` do VG
`vgextend lab /dev/sdc1 /dev/sdd1`

Roszerz LV do rozmiaru 300MB.
`lvextend /dev/lab/test -L 300M`

Sprawdź za pomocą `df -k`, że miejsce dostępne na LV się nie zwiększyło. Co prawda wolumin jest większy, ale system plików jeszcze o tym nie został poinformowany.

Wykonaj
`xfs_growfs /tmp/test`

Sprawdź komendą `df -k` , że wolne miejsce na woluminie zostało powiększone.

Woluminy ext2/ext3 można w nowszych linuxach powiększać i zmniejszać. Powiększanie i zmniejszanie może być wykonane na odłączonym woluminie komendą resize2fs. Wolumin podłączony może być tylko powiększany, służy do tego komenda ext2online.

Przygotuj PV `/dev/sdb1` do usunięcia (przenosząc zajęte bloki na inne PV w ramach grupy)
`pvmove /dev/sdb1`

Usuń `/dev/sdb1` z grupy lab
`vgreduce lab /dev/sdb1`

Ćwiczenie 8:
Celem tego ćwiczenia jest pokazanie złożonych „manewrów” jakie są potrzebne w sytuacji, gdy na systemie z działającym LVM chcemy podmieniać PV będące partycjami dyskowymi na macierze RAID 1 zapewniające niezawodność.

Aktualnie mamy VG lab utworzoną z PV /dev/sdc1 i /dev/sdd1. Dodatkowo dysponujemy partycjami /dev/sdb1 i /dev/sde1. Docelowo mamy mieć VG lab korzystającą z dwóch PV będących macierzami RAID1. Wszystko trzeba zrobić na działającym systemie.

Krok 1: w tym kroku połączysz /dev/sdb1 i /dev/sde1 w macierz RAID1 i podłączysz nową macierz do VG lab.

Utwórz macierz RAID 1 (`/dev/md0`) z partycji `/dev/sdb1` i `/dev/sde1` (zostaw typy partycji takie jakie są aktualnie - w testach to nie będzie przeszkadzać, normalnie powinny być zmienione na `0xfd`).

Zainicjalizuj PV `na /dev/md0` (`pvcreate`)
Podłącz `/dev/md0` do VG (`vgextend`)

Krok 2: Naszym celem jest teraz utworzenie macierzy z /dev/sdc1 i /dev/sdd1. Nie możemy tego zrobić od razu gdyż mamy za mało wolnego miejsca.W tym kroku usuniesz urządzenie /dev/sdc1 z VG lab i utworzysz na nim zdegradowaną macierz RAID1

Przygotuj i usuń z VG `/dev/sdc1` (`pvmove`, `vgreduce`)
Nie możemy usunąć `/dev/sdd1` bo jest za mało miejsca na VG.
Utwórz niekompletny RAID 1 (`/dev/md1`) z jednego urządzenia `/dev/sdc1` (drugim urządzeniem będzie „missing" oznaczający, że macierz ma być utworzona w trybie zdegradowanym)

```
mdadm -C -l1 -n2 /dev/md1 /dev/sdc1 missing
```

Zainicjalizuj PV na `/dev/md1` (`pvcreate`)
Podłącz `/dev/md1` do VG

Krok 3: W tej chwili mamy już wystarczająco dużo miejsca na VG żeby odłączyć /dev/sdd1 i dołączyć go do istniejącej macierzy

Przygotuj i usuń z VG `/dev/sdd1` (`pvmove`, `vgreduce`)

Podłącz `/dev/sdd1` do `/dev/md1`

```
mdadm -a /dev/md1 /dev/sdd1
```

Migracja została zakończona.

Laboratorium 5

- **Sieciowe systemy plików**

1. Ćwiczenia

Uwaga: Poniższe ćwiczenia należy wykonywać na DWÓCH wirtualnych maszynach AKiSO: AKISO-SERWER i AKISO-CLIENT (lub CENTOS 7 : SO-Client-CentOS7 i SO-Server-CentOS7)

Ćwiczenie 1:

Na systemie CENTOS7 zainstaluj pakiet nfs-utils na kliencie i serwerze

```
yum install nfs-utils (CENTOS7)
```

Komputer AKISO-SERWER udostępni katalog `/tmp/test` drugiemu komputerowi (AKISO-CLIENT). Za pomocą komedy `ifconfig` należy sprawdzić adresy obu komputerów (S.S.S.S oraz C.C.C.C).

Na serwerze należy teraz utworzyć plik `/etc/exports` zawierający linię:

```
/tmp/test  C.C.C.C(rw,sync)  (AKISO)
/tmp/test  C.C.C.C(rw,fsid=0,sync)  (CENTOS7)
```

gdzie `C.C.C.C` jest adresem IP klienta.

Uruchom serwer NFS poleceniami:

```
/etc/init.d/nfs start  (AKISO)
systemctl start nfs-server  (CENOS7)
iptables -F  (CENTOS7)
```

za pomocą polecenia `rpcinfo -p` sprawdź, czy działają wszystkie usługi nfs.

Na kliencie:

Utwórz katalog `/tmp/nfs`
Uruchom usługi NFS: (CENTOS7):

```
systemctl start nfs  (CENTOS7)
```

Podłącz katalog `/tmp/test` z serwera o adresie S.S.S.S w `/tmp/nfs`

```
mount -t nfs S.S.S.S:/tmp/test /tmp/nfs
```

Sprawdź, czy pliki tworzone na serwerze w katalogu /tmp/test są widoczne na drugim komputerze w katalogu `/tmp/nfs`.

Na serwerze utwórz użytkownika o nazwie aaa i uid=1001

```
useradd aaa -u 1001
```

Na kliencie utwórz użytkownika o nazwie BBB i uid=1001

Utwórz jakiś plik w `/tmp/test` i zmień jego właściciela na `aaa (chown aaa nazwa_pliku)`
Sprawdź, czy `aaa` jest jego właścicielem.
Sprawdź, kto jest właścicielem tego pliku na drugim komputerze.

Ćwiczenie 2:

Na systemie CENTOS7 zainstaluj pakiet cifs-utils na kliencie
```
yum install cifs-utils
```

Wyedytuj na serwerze plik `/etc/samba/smb.conf`

Dołóż **na końcu** tego pliku sekcję:

```
[test]
   path = /tmp/test
   public = no
   writeable = yes
```

Uruchom serwer samby komendą:
`/etc/init.d/smb start (AKISO)`
`setsebool -P samba_export_all_rw on (CENTOS7)`
`systemctl start smb (CENTOS7)`

`iptables -F (CENTOS7)`

Ustaw prawa 777 na katalogu `/tmp/test`

Ustaw hasło samby dla użytkownika root:
```
smbpasswd -a root
```

Na kliencie skorzystaj z programu smbclient (spróbuj użyć komend ls, get nazwa_pliku oraz quit):
`systemctl start smb (CENTOS7)`
```
smbclient '//S.S.S.S/test'
```

Podłącz na kliencie udział `//S.S.S.S/test`
W tym celu utwórz katalog `/tmp/samba` i wykonaj
```
mount -t cifs '//S.S.S.S/test' /tmp/samba
```

Skorzystaj z udziału `//S.S.S.S/test` udostępnianego przez wirtualną maszynę za pomocą Windows:

Kliknij Start->Uruchom, wpisz
```
\\S.S.S.S\test
```

Zwróć uwagę, że Windows korzysta z odwrotnego ukośnika (\) a nie z (/).

Laboratorium 6

- **Podstawy pisania programów w języku C, podstawowe operacje**

1. Zmienne shell-a

Interpreter poleceń shell ma zdefiniowany pewien zestaw zmiennych lokalnych. W odróżnieniu od zmiennych środowiskowych, zmienne shell-a nie są dziedziczone przez uruchamiane procesy (stąd mówimy, że są to zmienne lokalne). Zestaw ustawionych zmiennych shell-a można wyświetlić poleceniem `set` (zarówno w shell-u `bash` jak i `tcsh`).

W `bash`-u zmienne lokalne ustawia się pisząc polecenie postaci `NAZWA_ZMIENNEJ=WARTOŚĆ`. Zmienne lokalne można wyeksportować do zmiennych środowiskowych poleceniem `export`. Wówczas, jest to nadal ta sama zmienna, tylko widziana zarówno jako zmienna lokalna jak i środowiskowa.
W przypadku shell-a `tcsh` zmienne lokalne nie są powiązane ze środowiskowymi. Ustawia się je poleceniem `set NAZWA_ZMIENEJ=WARTOŚĆ`.

Przykłady:
- zmienne `HISTSIZE` (bash) lub `history` (tcsh) określają liczbę pamiętanych poleceń w historii.
- zmienne `PS1` (bash) lub `prompt` (tcsh) określają wygląd „znaczka zachęty" (początek wiersza poleceń)
- zmienne `TMOUT` (bash) lub `autologout` (tcsh) określają po jakim czasie nieaktywności użytkownika zostanie on automatycznie wylogowany. `TMOUT` jest podawany w sekundach, `autologout` w minutach.

Wartości zmiennych (lokalnych i środowiskowych) można wyświetlić poleceniem `echo $nazwa_zmiennej`
Symbole określające wygląd „znaczka zachęty" są różne dla bash i tcsh. Pełną listę można znaleźć w odpowiednich manualach. Przykładowe:

bash	tcsh	
`\w`	`%/`	katalog bieżący
`\t`	`%T`	bieżący czas w formacie 24-godzinnym
`\T`	`%t`	bieżący czas w formacie 12-godzinnym
`\!`	`%h`	numer bieżącego polecenia w historii
`\u`	`%n`	nazwa użytkownika
`\H` lub `\h`	`%M` lub `%m`	hostname
`$NAZWA`	`%$NAZWA`	zmienna środowiskowa lub lokalna shella o nazwie `NAZWA`

2. Podstawowe wskazówki odnośnie pisania i uruchamiania programów.

- Przyjmuje się, że plikom napisanym w języku C w systemie UNIX nadaje się rozszerzenie `.c` natomiast napisanym w C++ `.cpp`
- Popularne kompilatory to: `gcc` i `cc`. Najczęściej używana składnia przy kompilacji:
  `gcc plik_źródłowy -o plik_wynikowy`
  lub
  `cc plik_źródłowy -o plik_wynikowy`
  Nie podanie opcji `-o` oraz nazwy pliku wynikowego spowoduje, że zostanie utworzony domyślny plik wynikowy o nazwie `a.out`. Kompilatory mają szereg innych opcji, niektóre z nich będą poznawane na kolejnych zajęciach.
- Aby uruchomić program należy podać jego nazwę wraz ze ścieżką dostępu (względną lub bezwzględną). Dotyczy to również programów uruchamianych z katalogu bieżącego np.: `./program`
  Należy pamiętać, że jeśli podamy nazwę programu, który chcemy uruchomić bez podania ścieżki, będzie on poszukiwany w ścieżkach zapisanych w zmiennej `PATH` (w kolejności wystepowania ścieżek w zmiennej `PATH`).
  Przykładowo:
  `ls` – spowoduje zwykle uruchomienie `/usr/bin/ls` i wyświetlenie zawartości katalogu
  `./ls` – spowoduje uruchomienie programu o nazwie `ls` z katalogu bieżcego,
- Do uruchamianego programu można przekazać argumenty (parametry) wejściowe. Wymienia się je po nazwie programu, oddzielone spacjami:
  `./progr argument1 argument2 ...`
- Każdy program napisany poprawnie powinien zwracać jakąś wartość (kod wyjścia/zakończenia procesu). Jest to wartość zwracana przez funkcję `main()` w programie (patrz przykład 7). **Zgodnie z konwencją jest to wartość 0 gdy program zakończył się poprawnie (normalnie), lub dowolna inna wartość jeżeli**

wystąpiły błędy. Do programisty należy ustalenie jaka wartość będzie zwracana w różnych okolicznościach.
Kod wyjścia ostatnio zakończonego procesu przechowywany jest w zmiennej o nazwie `?` (pytajnik) i można go wyświetlić na terminalu wykonując polecenie:
`echo $?`

Można też wyświetlać kod zakończenia procesu w znaczku zachęty (prompt). Przykładowo:
`PS1='$? >'` (dla `bash`)
`set prompt="%? > "` (dla `tcsh`)

- Wyniki wykonania programu (wszelkie komunikaty z wyjątkiem komunikatów o błędach) program powinien zwracać na wyjście standardowe `stdout`. Do wyjścia standardowego `stdout` można pisać znanymi funkcjami `printf` lub `fprintf(stdout,"...",...)`.

  Komunikaty o błędach powinny być wysyłane na standardowe wyjście błędów `stderr` za pomocą funkcji `fprintf` albo funkcji `perror`. Funkcja `perror` automatycznie generuje komunikat opisujący błąd, który wystąpił. Błąd jest rozpoznawany na podstawie zawartości zmiennej `errno`, która będzie omówiona w dalszej części tekstu. Jako argument funkcji `perror` można podać dodatkowy tekst, który ma być wyświetlony. Wówczas na ekranie pojawia się „nasz tekst", znak „`:`" oraz automatycznie wygenerowany komunikat. Ponieważ funkcja `perror` tworzy gotowy komunikat o błędzie, jako dodatkowy tekst podaje się zwykle nazwę programu lub identyfikator procesu. Jest to wygodne zwłaszcza w sytuacji, kiedy pracuje równolegle wiele procesów, które wyrzucają komunikaty o błędach na tą samą konsolę – można się zorientować, od którego procesu pochodzi komunikat.

  Przykładowo:
  ```
  fprintf(stderr,"Komunikat\n");
  perror("Nasz komunikat");
  perror(argv[0]);
  ```

- Funkcje w języku C mogą zwracać wartości różnych typów (`char *`, `int`, `void` itp.). Dla funkcji bibliotecznych UNIXa często stosuje się pewne reguły umowne:
  - w przypadku funkcji zwracających wartości typu `char *` zwracany jest wskaźnik do miejsca w pamięci gdzie przechowywany jest znak (lub ciąg znaków) będący wynikiem wykonania funkcji. W przypadku błędu zwracany jest pusty wskaźnik – NULL.
  - funkcje zwracające wartości typu `int` często zwracają:
    – wartość `-1` w przypadku błędu
    – wartości różne od `-1` w przypadku poprawnego wykonania.

- W przypadku wystąpienia błędu większość funkcji bibliotecznych UNIXa wpisuje **kod błędu** do zmiennej globalnej **`errno`**. Kod błędu jest wartością typu `int`. Każda predefiniowana funkcja w C ma wyspecyfikowane kody błędów odpowiadające wystąpieniu różnego rodzaju błędów. W manualu do poszczególnych funkcji można znaleźć pełny opis możliwych błędów i związanych z nimi kodów. Kody błędu są podane w postaci stałych symbolicznych. Opis możliwych błędów i odpowiadających im stasłych symbolicznych można znaleźć w pliku pomocy `man errno`.

Przykład 7

Przedstawiono tylko fragment programu, zawierający przykład zastosowania kodów błędów.

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

int errno;

main(int argc,char *argv[])
{
int descr;
int error_code;
```

```
errno=0;

    descr=open(argv[1],O_RDWR);
    error_code=errno;
    if(descr<0)  /*zmienna descr ma wartosc -1 jezeli nie powiodlo sie
                 otwarcie pliku */
    {
       switch(error_code)
       {
         case EACCES: fprintf(stderr,"Blad: brak praw dostepu do pliku\n");
         case ENOENT: fprintf(stderr,"Blad: plik o podanej nazwie nie
istnieje\n");
       }

        return 2; /* wartość rozna od 0 bo program zakonczyl sie blednie */
    }
    else
    {
         /* ciag dalszy programu */
    }

close(descr);
return 0;   /* wartość 0 bo program zakonczyl sie poprawnie */
}
```

3. Pobranie argumentów (parametrów) wywołania programu.

Pobranie argumentów, z jakimi został uruchomiony proces jest możliwe, jeśli funkcja `main` została zadeklarowana z dwoma argumentami. Pierwszy (zwykle nazywany argc – argument counter) będzie przechowywać liczbę argumentów wywołania procesu, drugi (zwykle nazywany argv – argument values) będzie zawierać tablicę argumentów wywołania procesu. Poniżej przedstawiono dwa sposoby deklaracji tych zmiennych.

```
#include <stdio.h>

main(int argc, char *argv[])
{
      /*program*/
}

main(int argc, char **argv)
{
      /*program*/
}
```

W systemie UNIX każdy proces ma przynajmniej jeden argument wywołania. Jest nim nazwa uruchamianego programu. Oznacza to, że zmienna `argc` ma wartość większą lub równą `1` zaś `argv[0]` przechowuje nazwę programu. Długość tablicy `argv` to zawsze `argc+1`. Ostatni element tej tablicy (`argv[argc]`) ma zawsze wartość `NULL`. Ilustruje to przykład 8.

Przykład 8

wywołanie	wartości zmiennych	
`> ./progr`	`argc = 1`	`argv[0] = ./progr` `argv[1] = NULL`

`> ./progr arg1 arg2 ppp xyz`	`argc = 5`	`argv[0] = ./progr`
		`argv[1] = arg1`
		`argv[2] = arg2`
		`argv[3] = ppp`
		`argv[4] = xyz`
		`argv[5] = NULL`

4. Pobieranie zmiennych środowiskowych.

Pełną listę zmiennych środowiskowych można odczytać w programie na dwa sposoby:

- Deklarując dodatkowy argument funkcji `main`

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

  W tym przypadku zadeklarowanie zmiennych `argc` oraz `argv` jest konieczne niezależnie od tego, czy przewiduje się w programie korzystanie z argumentów wywołania czy nie.
  W tablicy `envp` pod kolejnymi indeksami przechowywane będą wszystkie zmienne środowiskowe jako ciągi znaków postaci: `NAZWA_ZMIENNEJ=WARTOSC`
  Przykładowo:
  `HOST=pluton.kt.agh.edu.pl`
  Ostatni element tablicy `envp` ma zawsze wartość `NULL`.

- Korzystając ze zmiennej zewnętrznej `environ`. Sposób przechowywania zmiennych środowiskowych oraz korzystanie z nich jest analogiczne jak w poprzednim przypadku. Nie ma konieczności deklarowania zmiennych `argc` oraz `argv`.

  ```
  main()
  {
        extern char **environ;

        /*program*/
  }
  ```

Można również pobrać wartość wybranej zmiennej środowiskowej. Służy do tego funkcja:

```
char *getenv(char *name);           <stdlib.h>
```

Argumentem funkcji jest wskaźnik do zmiennej przechowującej nazwę żądanej zmiennej środowiskowej.
Funkcja zwraca wskaźnik do wartości zmiennej środowiskowej.

Można również tworzyć własne zmienne środowiskowe lub modyfikować istniejące. Służy do tego funkcja:

```
int putenv(char *name);       <stdlib.h>
```

Argumentem funkcji jest wskaźnik do zmiennej przechowującej wartość w postaci:
`nazwa_zmiennej=wartość_zmiennej`.

5. Otwieranie, tworzenie i zamykanie pliku.

a) Otwieranie

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

int open(const char *path, int oflag [, int mode ]);
```

Funkcja zwraca deskryptor pliku lub -1 w przypadku błędu.
Do zmiennej `errno` zapisywany jest kod błędu.

Pierwszym argumentem funkcji jest ścieżka dostępu do otwieranego pliku. Drugim jest flaga oznaczająca tryb otwarcia pliku. Najważniejsze flagi:

O_RDONLY -otwórz tylko do czytania

O_WRONLY -otwórz tylko do pisania
O_RDWR -otwórz do pisania i czytania
O_APPEND -dopisywanie na końcu pliku
O_TRUNC -jeżeli plik istnieje to jego długość jest zmniejszana do 0.
O_CREAT -jeżeli plik nie istnieje to ma być utworzony
O_EXCL -jeżeli użyto O_CREAT a plik istnieje to funkcja zgłosi błąd

Jeśli plik, który próbujemy otworzyć nie istnieje i nie zastosowano flagi O_CREAT to funkcja `open` zgłosi błąd. Zastosowanie tej flagi spowoduje utworzenie i otwarcie pliku. Jeżeli ustawiono jednocześnie dwie flagi: O_CREAT i O_EXCL to:

- jeżeli plik istnieje funkcja zgłosi błąd,
- jeśli plik nie istnieje, to zostanie utworzony i otwarty.

Jeśli zastosowano flagę O_CREAT konieczne jest określenie, jakie prawa dostępu mają być nadane nowo utworzonemu plikowi. Służy do tego opcjonalny trzeci argument `mode`. Należy pamiętać, że żądane prawa dostępu są dodatkowo „filtrowane" przez maskę praw dostępu. Przykładowo, jeżeli zażądano nadania prawa „w" dla grupy a maska praw dostępu tego zabrania, to prawo to nie będzie nadane.

b) Tworzenie

Do utworzenia nowego pliku można również użyć funkcji `creat`:

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

int creat(char *pathname, int mode);
```

Jeżeli zakończy się powodzeniem zwróci deskryptor pliku. Plik będzie otwarty do pisania.
W razie wystąpienia błędu funkcja zwraca -1

c) Zamykanie

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

int close(int descr);
```

Zwraca 0 jeżeli wykonane poprawnie lub -1 w przypadku błędu.

Dowiedz się samodzielnie do czego słuzy funkcja języka C `mknod`. Czym różni się od funkcji `creat`.

W systemach UNIX-owych do obsługi plików można też używać funkcji posługujących się wskaźnikami na plik: `FILE *fopen(...)`, `FILE *fclose(...)` itp. Funkcje te są wprawdzie bardziej powszechne i uniwersalne (są standardowymi funkcjami bibliotecznymi dostępnymi w każdej wersji języka C), lecz nie obsługują mechanizmów specyficznych dla UNIX-a (np. prawa dostępu, typ pliku, linki itp.). Z tego względu, w przypadku pisania programów pod UNIX-a częściej używa się funkcji z rodziny `open`, `close` itd.

6. Ustalenie praw dostępu do pliku

Prawa dostępu do pliku przechowywane są w zmiennych typu `int`. Aby można było je łatwo ustawiać i interpretować najlepiej jest przedstawiać liczbę w zapisie ósemkowym. W języku C każda liczba zaczynająca się od `0` jest rozumiana przez kompilator jako liczba ósemkowa (podobnie jak liczby zaczynające się od `0x` są liczbami w kodzie heksadecymalnym, patrz przykład 9). Przykładowo liczba `0751` oznacza prawa dostępu: `rwxr-x--x`. Dla zwiększenia przejrzystości kodu programu zdefiniowano ponadto szereg stałych symbolicznych określających poszczególne prawa dostępu. Przedstawia je poniższa tabela. Oczywiście można tworzyć kombinacje tych praw łącząc stałe symboliczne operatorem sumy logicznej.

Stała symboliczna	Wartość	Opis
S_ISUID	04000	Set user ID on execution.
S_ISGID	02000	Set group ID on execution.
S_ISVTX	01000	Save text image after execution.
S_IRWXU	00700	Read, write, execute by owner.
S_IRUSR	00400	Read by owner.
S_IWUSR	00200	Write by owner.
S_IXUSR	00100	Execute (search if a directory) by owner.
S_IRWXG	00070	Read, write, execute by group.
S_IRGRP	00040	Read by group.
S_IWGRP	00020	Write by group.
S_IXGRP	00010	Execute by group.
S_IRWXO	00007	Read, write, execute (search) by others.
S_IROTH	00004	Read by others.
S_IWOTH	00002	Write by others.
S_IXOTH	00001	Execute by others.

Przykład 9

Reprezentacja liczb w języku C:
`1938` - liczba dziesiętna
`0275` - liczba w kodzie ósemkowym
`0x2f4a` - liczba w kodzie heksadecymalnym

Przykład 10

Tworzymy plik funkcją `creat`. Prawa dostępu `rw-r-----` można nadać temu plikowi na dwa sposoby:

```
creat(”nowy_plik”,0640);
creat(”nowy_plik”, S_IRUSR | S_IWUSR | S_IRGRP);
```

7. Informacje o pliku zawarte w węźle pliku

Zdefiniowane są dwie funkcje systemowe pozwalające odczytać informacje o pliku zapisane w jego węźle (i-node).

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

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

`path` — ścieżka do pliku,
`fildes` — deskryptor pliku,

Jeżeli wykonanie funkcji zakończy się sukcesem to zwróci ona wartość 0. W przypadku przeciwnym zwróci -1. Oprócz tego wynikiem poprawnego wykonania funkcji jest wypełnienie struktury typu `stat` wskazywanej przez wskaźnik `buf`. Definicja tej struktury jest następująca:

```
struct stat {
    mode_t   st_mode;     /* File mode (see mknod(2)) */
    ino_t    st_ino;      /* Inode number */
    dev_t    st_dev;      /* ID of device containing */
                          /* a directory entry for this file */
    dev_t    st_rdev;     /* ID of device */
                          /* This entry is defined only for */
                          /* char special or block special files */
    nlink_t  st_nlink;    /* Number of links */
    uid_t    st_uid;      /* User ID of the file's owner */
    gid_t    st_gid;      /* Group ID of the file's group */
    off_t    st_size;     /* File size in bytes */
    time_t   st_atime;    /* Time of last access */
    time_t   st_mtime;    /* Time of last data modification */
    time_t   st_ctime;    /* Time of last file status change */
};
```

Pierwsza zmienna tej struktury `st_mode` jest liczbą, która wyrażona w systemie ósemkowym reprezentuje typ pliku oraz prawa dostępu (czyli słowo trybu dostępu – patrz zajęcia 1) w postaci:

`0FFSUGO`

gdzie FF - typ pliku, S - bity s i t, UGO - prawa dostępu odpowiednio dla użytkownika, grupy i pozostałych.

Wartości związane z poszczególnymi typami plików oraz odpowiednie stałe symboliczne przedstawiono poniżej:

```
S_IFMT          0170000        TYP PLIKU
S_IFREG         0100000        ZWYKLY
S_IFDIR         0040000        KATALOG
S_IFCHR         0020000        SPEC. ZNAKOWY
S_IFBLK         0060000        SPEC. BLOKOWY
S_IFLNK         0120000        DOWIAZANIE SYMBOLICZNE
S_IFSOCK        0140000        GNIAZDO
S_IFIFO         0010000        KOLEJKA FIFO
```

Dowiedz się samodzielnie czym różnią się funkcje `stat` i `lstat`.

Przykład 11

Jeżeli `st_mode` ma wartość 0060755 to oznacza, że jest to plik specjalny blokowy (6) i nadano prawa `rwxr-xr-x` (755).

Wyróżnia się trzy czasy modyfikacji związane z każdym węzłem. Czas dostępu do pliku jest zmieniany przy każdym odczycie zawartości pliku. Czas modyfikacji pliku jest zmieniamy przy każdej zmianie zawartości pliku. Czas modyfikacji stanu pliku zmieniany jest wtedy, gdy zmianie ulegają informacje zawarte w węźle (zawartość pliku może pozostać niezmieniona. Wpływ przykładowych operacji na czasy modyfikacji przedstawiono w poniższej tabeli.

	zapis – np. przekierowanie	`touch`	zamiana za pomocą edytora	odczyt np. `cat`, `less`	zmiana praw `chmod`	utworzenie/usunięcie dowiązania twardego	kopiowanie `cp`	zmiana nazwy `mv`
dostępu		x	x	x			x	
modyfikacji	x	x	x					
zmiany stanu pliku	x	x	x		x	X		x

8. Zapis i odczyt z pliku.

Do zapisu i odczytu pliku stosuje się funkcje `read` i `write`.

```
#include <unistd.h>

int read(int descr, char *buff, unsigned int nbytes);
```

Pierwszym argumentem jest deskryptor pliku, drugim wskaźnik do bufora, w którym będą przechowywane wczytane znaki (bajty), a trzecim liczba bajtów, które powinny być wczytane (w jednym wywołaniu funkcji `read`). Liczba wczytywanych bajtów `nbytes` nie może być większa niż rozmiar bufora.

Jeżeli funkcja wykona się poprawnie zwraca liczbę wczytanych bajtów. W przypadku napotkania końca pliku funkcja zwraca 0. W przypadku błędu funkcja zwraca -1.

```
#include <unistd.h>

int write(int descr, char *buff, unsigned int nbytes);
```

Funkcja zwraca liczbę zapisanych bajtów lub -1 w przypadku błędu.

9. Pytania kontrolne

1. Co to są zmienne lokalne shell-a? Jakie znasz przykłady zmiennych shell-a?
2. Jaka jest domyślna nazwa dla pliku wynikowego utworzonego przy kompilacji za pomocą `cc` lub `gcc`?
3. Czym się różnią następujące polecenia:
   ```
   ./progr
   progr
   ```
4. Co to jest kod zakończenia procesu?
5. Podaj różnice w działaniu funkcji do generowania komunikatów o błędach: `perror` i `fprintf`.
6. Do czego służy zmienna `errno`?
7. Jak sprawdzić czy wywołana funkcja biblioteczna wykonała się poprawnie?
8. Jakie zmienne przechowują argumenty wywołania programu?
9. Jaką liczbę argumentów będzie miał program test wywołany w następujący sposób: `./test 3`
10. Jakie znasz sposoby dostępu do zmiennych środowiskowych?
11. W jakim systemie zapisane są liczby: 0345, 785, 0x451
12. Czym różnią się funkcje `stat` i `lstat`?
13. Jakie czasy modyfikacji są związane z każdym węzłem pliku?
14. Jaki czas/czasy modyfikacji jest zmieniany, gdy tworzymy nowe dowiązanie twarde do pliku?
15. Jaki czas/czasy modyfikacji są zmieniane, kiedy odczytamy zawartość pliku?
16. Czym różni się funkcja `mknod` od funkcji `creat`.

10. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl`
Należy je wykonywać (tworzyć pliki, katalogi) w swoich katalogach domowych.
Przykładowe programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB06`

Ćwiczenie 1:

- Ustaw „znak zachęty" (prompt) tak, aby wyświetlał bieżący czas a po spacji bieżący katalog np.: `12:08:02 /export/home >`

- Ustaw „znak zachęty" (prompt) tak, aby wyświetlał kod wyjścia ostatnio zakończonego procesu. Przetestuj wykonując np. polecenie `ls` z nazwą istniejącego, a później nieistniejącego pliku.

- ustaw czas automatycznego wylogowania na 1 minutę. Poczekaj, aby sprawdzić czy zadziała

Ćwiczenie 2:

Za katalogu `~rstankie/stud/LAB06` skopiuj na swoje konto plik `arg.c`. Obejrzyj kod programu. Jakich wyników działania programu się spodziewasz?
Skompiluj program tak, aby kod wynikowy znalazł się w pliku o nazwie `a.out`. Skompiluj ponownie tak, aby kod wynikowy znalazł się w pliku o nazwie `arg`.

Uruchom program `arg` bez argumentów. Zobacz wyniki.

Uruchom program `arg` z kilkoma argumentami. Zobacz wyniki.

Ćwiczenie 3:

Skopiuj na swoje konto plik o nazwie `opcl.c`. Obejrzyj kod programu. Jakich wyników działania programu się spodziewasz? Skompiluj program tak, aby kod wynikowy znalazł się pliku o nazwie `opcl`.

Utwórz plik tekstowy o nazwie `aa` np. poleceniem `ls > aa`.
Wyświetl zawartość pliku `aa` za pomocą programu `opcl`.

Spróbuj wyświetlić zawartość nieistniejącego pliku o nazwie np. `bb`. Sprawdź jaki jest kod zakończenia tego procesu (np. `echo $?`)

Zmodyfikuj program `opcl.c` tak, aby w przypadku braku pliku o podanej nazwie nie był zgłaszany błąd, lecz tworzony był nowy plik z prawami dostępu `r-s-w--wx`. Sprawdź czy na pewno program działa poprawnie!

Ćwiczenie 4:

Skopiuj na swoje konto pliki `type.c` oraz `stat.c`. W obu programach użyto funkcję stat do wydobycia informacji zawartych w węźle pliku. Program `type` mówi, jaki jest typ pliku, którego nazwę podano jako argument, zaś program `stat` wyświetla na ekranie szereg informacji z węzła pliku, którego nazwę podano jako argument.

Obejrzyj kod programów.

1. Za pomocą programu `type` sprawdź jakiego typu są pliki `aa`, `/dev/null`, `/dev/hda`, `/dev/log`
2. Utwórz plik dowiązanie symboliczne o nazwie `abc` do dowolnego pliku zwykłego w swoim katalogu. Sparadź programem `type` jaki jest typ pliku `abc`. Dlaczego nie „dowiązanie symboliczne"? Zmodyfikuj progam type.c aby uzyskać poprawną infomację o pliku `abc`.

3. Utwórz pusty plik tekstowy o nazwie `test` poleceniem `touch`. Wyświetl informacje o tym pliku za pomocą programu `stat`.
4. Zwróć uwagę na liczbę dowiązań, identyfikatory właściciela pliku, rozmiar pliku oraz czasy modyfikacji.

5. Zmodyfikuj plik za pomocą edytora `vi`. Wpisz dowolny tekst. Sprawdź ponownie informacje o pliku. Jakie czasy się zmieniły? Czy czas odczytu jest równy czasowi zapisu? Dlaczego?
6. Utwórz nowe dowiązanie do pliku `test` o nazwie `test2`. Co się zmieniło w węźle?
7. Skopiuj plik `test` do pliku `test_copy`. Sprawdź ponownie informacje o pliku `test`. Jaki czas zmienił się?
8. Dopisz na końcu pliku wyniki wykonania polecenia `date` (poprzez przekierowanie). Sprawdź ponownie informacje o pliku `test`.
9. Wykonaj polecenie `touch test`. Jakie teraz są czasy modyfikacji?

Ćwiczenie 5:

Napisz program, który będzie wyświetlał wartości zmiennych środowiskowych wg poniższych reguł:

- jeżeli nie podano żadnych argumentów wyświetlane są wszystkie zmienne środowiskowe
- jeżeli podano jakieś argumenty, to są one traktowane jako nazwy zmiennych, których wartości chcemy wyświetlić. Np. wywołanie `./zmienne PATH HOST LL` powinno wyświetlić na ekranie wartości zmiennych PATH, HOST i LL. Jeżeli zmienna o podanej nazwie nie istnieje powinien pojawić się stosowny komunikat.

Program należy napisać za zachowaniem omówionych w materiałach reguł.

Dla chętnych: zobacz inne programy w katalogu `~rstankie/stud/LAB06`

Laboratorium 7

- **Procesy, sygnały**

1. Wprowadzenie

Procesem jest każdy program, który został uruchomiony i jest wykonywany. System operacyjny UNIX jest systemem wieloprocesowym, czyli możliwe jest wykonywanie wielu procesów jednocześnie. W praktyce, jeden procesor może obsługiwać w danej chwili tylko jedno zadanie (jeden proces). Każdy z działających procesów otrzymuje na krótki czas dostęp do procesora. Dzięki temu zapewniona jest pozorna równoległość wykonywania procesów.

a) Lista procesów

Wyświetlenie listy procesów w systemie umożliwia polecenie `ps`. Użyte bez żadnych opcji wyświetla tylko informacje o procesach użytkownika, który je wykonał. Poszczególne opcje pozwalają na wyświetlenie różnorodnych informacji o procesie. Przedstawiono tylko niektóre z nich:

- `ps -f` dodatkowe parametry procesu
- `ps -l` dodatkowe parametry procesu
- `ps -e` wszystkie procesy w systemie
- `ps –u username` procesy o identyfikatorze efektywnym użytkownika `username`
- `ps –o format` wyświetla parametry procesu określone przez `format` (przykład 1)

Wybrane parametry procesu:

- `UID` - efektywny identyfikator użytkownika
- `USER` - nazwa użytkownika (efektywnego)
- `RUID` - rzeczywisty identyfikator użytkownika
- `RUSER` - nazwa użytkownika (rzeczywistego)
- `GID` - efektywny identyfikator grupy
- `GROUP` - nazwa grupy (efektywna)
- `RGID` - identyfikator grupy (rzeczywisty)
- `RGROUP` - nazwa grupy (rzeczywista)
- `PID` - identyfikator procesu - numeryczny
- `PPID` - identyfikator procesu rodzica – numeryczny
- `PGID` -identyfikator grupy procesów
- `C` - czas CPU zużyty przez system (*clock ticks*)
- `STIME` - czas startu w systemie
- `PRI` - priorytet procesu (im mniejsza liczba tym wyższy priorytet)
- `NI` - liczba użyta do obliczenia priorytetu (*nice number*)
- `ADDR` - numer segmentu stosu procesu
- `SZ` - rozmiar obrazu pamięci procesu [1kB] (*core image*)
- `TTY` - terminal, z którego uruchomiono proces (− jeśli brak, `?` nieznany)
- `TIME` - całkowity czas wykonywania procesu
- `CMD` - nazwa komendy (wywołania procesu)
- `SSIZ` - rozmiar stosu jądra systemu
- `SIZE` - rozmiar wirtualnej sekcji pamięci danych
- `RSS` - rozmiar rzeczywistej pamięci procesu
- `%CPU` - procent wykorzystania czasu CPU od początku pracy procesu
- `%MEM` - procent pamięci zajętej przez proces

Więcej na temat poszczególnych identyfikatorów procesu będzie wyjaśnione w dalszej części materiałów.

Przykład 12

```
> ps -o user,group,ruser,rgroup,pid,ppid,pgid,fname

    USER    GROUP    RUSER   RGROUP   PID  PPID  PGID COMMAND
rstankie    staff rstankie    staff 20540 20535 20540 tcsh
rstankie    staff rstankie    staff 24611 20540 24611 ps
```

d) Uruchamianie procesów

Uruchomienie pliku wykonywalnego (programu) powoduje utworzenie procesu. Proces może zostać uruchomiony:

- na pierwszym planie (np.: `./program`), wówczas przez czas działania programu nie ma dostępu do wiersza poleceń terminala;
- w tle (np.: `./program &`), wówczas proces działa niejako na drugim planie nie blokując dostępu do terminala (można wykonywać inne polecenia, uruchamiać inne procesy)

c) Usuwanie procesów

Działanie procesu można przerwać w dowolnym momencie przez wysłanie do niego odpowiedniego sygnału (sygnały będą omówione dalej). Do wysyłania sygnałów służą:

- polecenie `kill [-signal] pid`
- pewne kombinacje klawiszy np.: `^C` – wysłanie sygnału INT (interrupt) do procesu – przerywa działanie bieżącego procesu działającego na pierwszym planie

d) Zatrzymanie procesu

Działający proces może zostać zatrzymany (uśpiony) na pewien czas. Proces wówczas nie jest wykonywany (nie jest mu przydzielany czas procesora). Zatrzymanie procesu następuje poprzez wysłanie do niego sygnału TSTP lub STOP. Sygnał TSTP można wysłać do procesu przez naciśnięcie kombinacji klawiszy `^Z` (pod warunkiem, że proces działa na pierwszym planie).

e) Obudzenie procesu uśpionego

Uśpiony proces może zostać ponownie przywrócony do działania. Można tego dokonać wysyłając do procesu sygnał `CONT`. Można też użyć jedno z dwóch poleceń:

- `fg` – przywrócenie do działania na pierwszy plan
- `bg` – przywrócenie do działania w tle.

Ponadto, poleceniem `fg` można przenieść na pierwszy plan proces działający w tle.

Polecenie `jobs` wyświetla listę procesów uśpionych oraz działających w tle.

Przykład 13

Polecenia `jobs`, `fg` i `bg`.

```
> jobs
[1]  - Running                 ns symul.tcl
[2]  + Suspended               vi dtf_g.c
```

Proces `ns symul.tcl` działa w tle, zaś proces `vi dtf_g.c` jest uśpiony.

Obudzenie w tle procesu `vi dtf_g.c`.

```
> bg %2
```

Przeniesienie procesu `ns symul.tcl` na pierwszy plan.

```
> fg %1
```

2. Identyfikatory użytkownika procesu i grupy użytkowników procesu

Każdy plik na dysku ma ściśle określonego właściciela. Plik wykonywalny będący własnością jednego użytkownika może zostać uruchominy przez innego uzytkownika, który staje się właścicielem procesu.
Każdy proces ma zestaw identyfikatorów, które pozwalają określić jego właściciela oraz uprawnienia. Wyróżnia się identyfikatory rzeczywiste oraz efektywne (zwane też obowiązującymi).

Identyfikatory rzeczywiste zawsze określają użytkownika, który uruchomił proces oraz jego grupę. Identyfikują więc właściciela procesu.
Identyfikatory efektywne służą do określenia uprawnień z jakimi proces działa w systemie. Są one istotne między innymi w sytuacji, gdy proces próbuje wykonać jakąś operację (przeczytać lub zapisać plik, uruchomić inny program, przeczytać katalog itp.). Identyfikatory efektywne mogą być równe rzeczywistym (jeśli uruchamiany program nie miał ustawionego żadnego bitu „s") lub podmienione na odpowiednie identyfikatory właściciela uruchamianego pliku. Bit „s" dla właściciela pliku (`rws`r-xr-x) oznacza, że proces będzie miał ustawiony efektywny identyfikator użytkownika równy identyfikatorowi właściciela pliku. Bit „s" dla grupy pliku (rwxr-`s`r-x) oznacza, że proces będzie miał ustawiony efektywny identyfikator grupy równy identyfikatorowi grupy pliku. Bity „s" dla właściciela i grupy mogą oczywiście być ustawione jednocześnie.

Uwaga!

Zwykle identyfikatory rzeczywiste oznacza się `uid` oraz `gid`, zaś efektywne: `euid` oraz `egid`. Jednakże w formacie polecenia `ps`, identyfikatory rzeczywiste to `ruid` oraz `rgid`, zaś efektywne to `uid` oraz `gid`.

Przykład 14

Użytkownik `adam` (`uid=200`) jest członkiem grupy `staff` (`gid=35`). Jest on właścicielem programu `test`.

Użytkownik `tom` (`uid=314`) należący do grupy `stud` (`gid=40`) uruchomił ten program.

a) Jeżeli plik `test` miał prawa dostępu: `rwxr-xr-x` (nie ustawiono bitów „s") wówczas identyfikatory procesu są równe identyfikatorom użytkownika, który proces ten uruchomił:

```
uid=euid=314
gid=egid=40
```

b) Jeżeli plik test miał prawa dostępu: rw`s`r-xr-x wówczas identyfikatory procesu są następujące:

```
uid=314
euid=200
gid=egid=40
```

Uruchomiony proces działa więc z uprawnieniami właściciela pliku `test`, a nie użytkownika, który go uruchomił (uprawnienia grupy nie zostały zmienione). „Efektywnie" proces `test` będzie traktowany tak jakby uruchomił go użytkownik `adam` i grupa `stud`. Działa z prawami użytkownika `adam` i grupy `stud`.

Funkcje w języku C służące do uzyskania rzeczywistych i efektywnych identyfikatorów użytkownika i grupy:

- Rzeczywisty identyfikator użytkownika procesu:

```
int getuid()
```

- Efektywny identyfikator użytkownika procesu:

```
int geteuid()
```

- Rzeczywisty identyfikator grupy:

```
int getgid()
```

- Efektywny identyfikator grupy:

```
int getegid()
```

3. Identyfikatory procesu, rozwidlanie procesów

„fork bomba" – dodać opis i przykład

W systemie UNIX wszystkie procesy posiadają swoje identyfikatory PID (*Process Identifier*). Identyfikator jest unikalny. W danej chwili nie ma w systemie dwóch działających procesów o tym samym PID.

Wszystkie procesy (z wyjątkiem procesu `init`) tworzone są przy pomocy tego samego mechanizmu – rozwidlania procesów (wywołanie funkcji systemowej `fork`). Jeżeli proces wywoła funkcję `fork`, wówczas w systemie powstanie nowy proces będący jego kopią. Proces, który wykonał funkcję `fork` nazywany jest

procesem macierzystym (rodzicem) a proces nowopowstały – potomkiem (dzieckiem). Dlatego też, dla każdego procesu zawsze można określić proces, który go utworzył (z wyjątkiem procesu `init`). Proces `init` jest pierwszym procesem powstającym przy starcie systemu.

W momencie utworzenia proces otrzymuje swój unikalny identyfikator PID przydzielony przez system. Nie da się przewidzieć wartości PID (jedynie proces `init` ma zawsze taki sam numer PID równy 1). Każdy proces otrzymuje też identyfikator PPID (*Parent Process Identifier*), który jest równy identyfikatorowi PID jego rodzica. W ten sposób można jednoznacznie określić rodzica danego procesu.

Z każdym procesem związany jest jeszcze identyfikator grupy procesów GPID (*Group Process Identifier*). Procesy należące do jednej grupy mają taki sam GPID. Proces, którego GPID jest równy jego PID-owi zwany jest nadzorcą grupy procesów. Domyślnie procesy dziedziczą GPID od swojego procesu macierzystego. W każdej chwili proces może sam siebie uczynić nadzorcą grupy procesów (wydzielić własną grupę procesów). Odpowiednie mechanizmy w shell'u zapewniają, że proces uruchomiony z shell'a nie dziedziczy jego GPID lecz zakłada własną grupę procesów.

Do uzyskania wartości PID, PPID oraz GPID procesu służą następujące funkcje języka C:
```
pid_t getpid();
pid_t getppid();
pid_t getpgrp();
```
Do rozwidlania procesów służy funkcja:
```
pid_t fork();
```

Wszystkie powyższe funkcje zadeklarowane są w nagłówku `<unistd.h>`

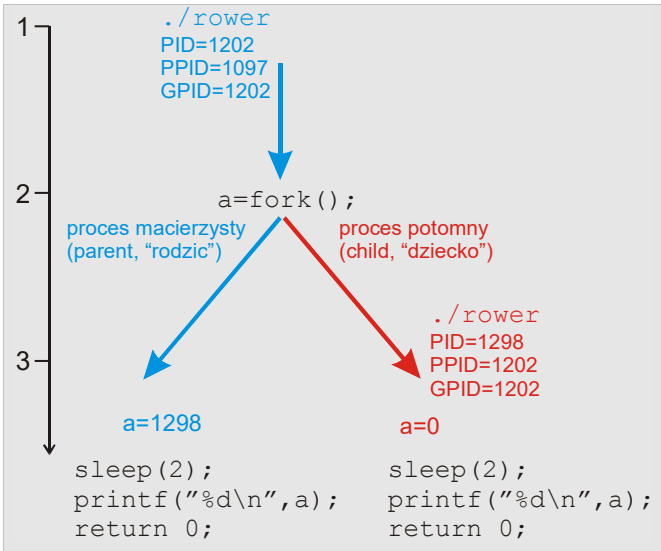
Funkcja `fork` zwraca wartość zero w procesie potomnym, natomiast w procesie tworzącym (macierzystym) zwracana jest wartość identyfikatora PID utworzonego procesu (w ten sposób proces macierzysty zna PID swojego potomka). W przypadku, gdy nie udało się utworzyć procesu potomnego zwracana jest wartość -1, a kod błędu jest wpisywany do globalnej zmiennej `errno`.

Przykład 15

Przykład przedstawia mechanizm rozwidlania procesów i przydzielania identyfikatorów PID, PPID. Dla porządku na rysunku umieszczono również wartości GPID.
Załóżmy, że kod programu `rower.c` jest następujący (pominięto biblioteki):

```
main(){
      int a;
      a=fork();

      sleep(2);
      printf("%d\n",a);
      return 0;
}
```

Kolejne etapy wykonania procesu wyglądają nastepująco:



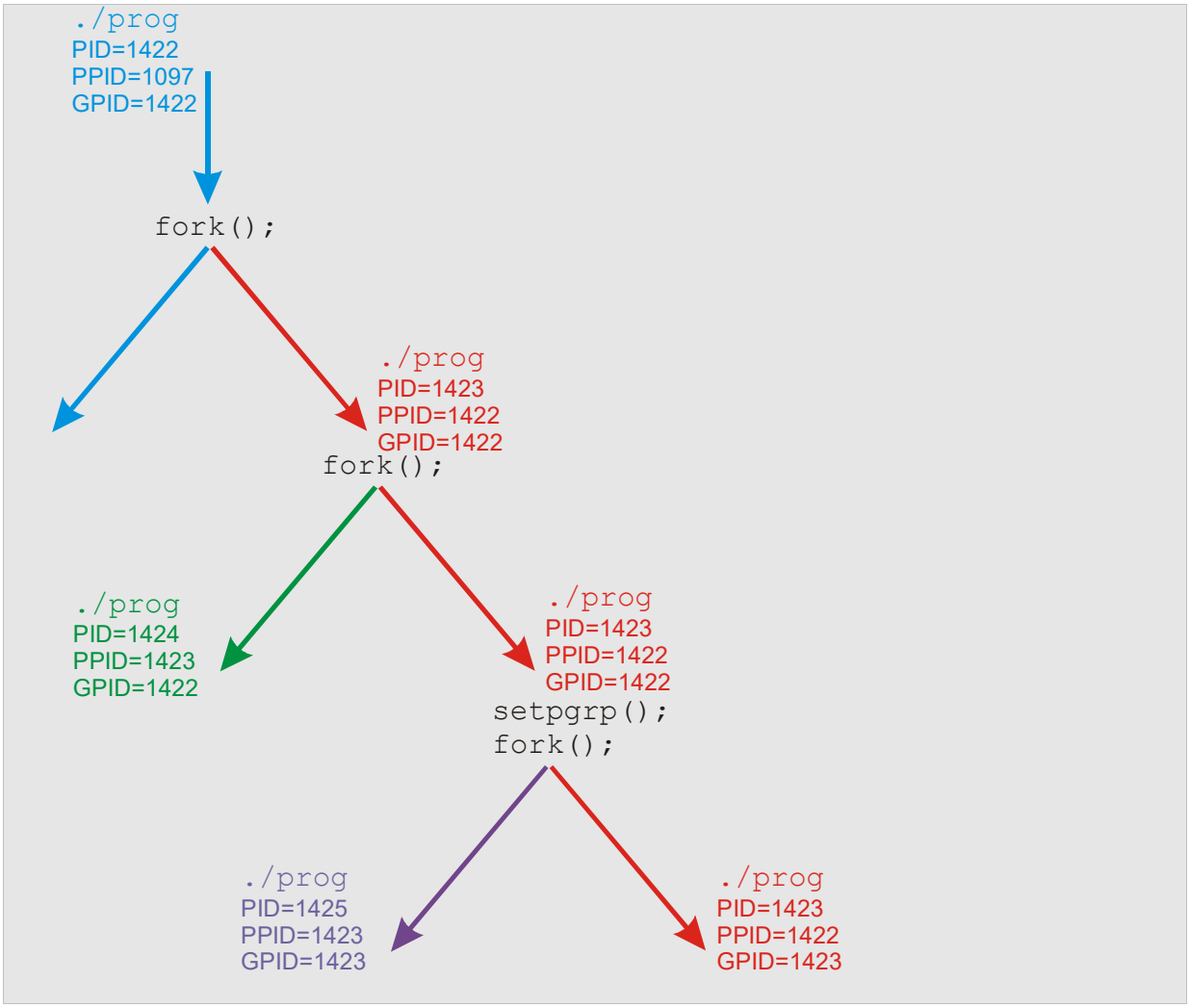
- W chwili (1) uruchomiono program `./rower`. Interpreter poleceń (shell) miał PID równy 1097. Procesowi `./rower` przydzielony został PID=1202, zaś jego PPID jest równy PID-owi procesu shell'a – 1097.
- W chwili (2) proces `./rower` wykonał funkcję `fork()` i utworzył w systemie swoją nową kopię. Nowy proces zwany jest procesem potomnym, zaś proces, który wywołał funkcję `fork()` procesem macierzystym. Nowo powstałemu procesowi został przydzielony PID równy 1298. Jego PPID jest równy PID-owi procesu macierzystego, czyli 1202.
- Funkcja `fork()` zwróciła w procesie macierzystym wartość 1298 (PID potomka), zaś w procesie potomnym wartość 0. Oznacza to, że w chwili (3) wartości zmiennej a w obu procesach będą odpowiednio 1298 i 0.

Obydwa procesy mają ten sam kod i są niejako na tym samym etapie wykonywania swojego kodu. Kolejną operacją, jaką wykonają obydwa procesy będzie funkcja `sleep(2)`. Następnie oba procesy wykonają funkcję `printf` i zakończą swoje działanie.

Przykład 16

Przykład przestawia mechanizm wydzielenia nowej grupy procesów.

Proces „czerwony" (PID=1423) należy do grupy procesu „niebieskiego" (PID=1422). Utworzył potomka „zielonego" (PID=1424) po czym uczynił siebie nadzorcą grupy. Należy zwrócić uwagę, że utworzony wcześniej potomek „zielony" nadal należy do grupy procesu „niebieskiego". Natomiast kolejne procesy tworzone potomne procesu „czerwonego" będą już należały do jego grupy (GPID=1423).



4. Podmiana kodu

Mechanizm rozwidlania procesów fork powoduje utworzenie tylko kopii procesu, co daje ograniczone możliwości tworzenia różnorodnych procesów. Aby było możliwe uruchamianie innych programów konieczny jest jeszcze mechanizm podmiany kodu procesu.
Przykładowo, gdy uruchamiamy program `ls`, w pierwszej chwili powstaje kopia procesu naszego shell'a, której kod jest następnie podmieniany na kod programu `ls`.
Każdy proces może w dowolnym momencie podmienić swój kod na inny (czyli zupełnie zmienić swoje własności i funkcje). **W momencie podmiany kodu nie zmieniają się identyfikatory procesu**.

Dostępna jest rodzina funkcji w języku C służących do podmiany kodu:

```
#include <unistd.h>

int execl(const char *path, const char  *arg0, ..., const char *argn, char
* /*NULL*/);

int execv(const char *path, char *const argv[]);

int execle(const char *path, const char *arg0, ..., const char *argn, char
* /*NULL*/, char *const envp[]);

int execve(const char *path, char *const argv[], char *const envp[]);

int execlp(const char *file, const char  *arg0, ..., const char *argn, char
* /*NULL*/);

int execvp(const char *file, char *const argv[]);
```

5. Oczekiwanie na zakończenie procesów potomnych

Procesy powinny kończyć swoje działanie w kolejności odwrotnej niż powstawały. Proces macierzysty nie powinien zakończyć się wcześniej niż jego procesy potomne. Jednakże jest to możliwe. Jeśliby się tak stało, PPID w procesie potomnym utraci ważność. Istnieje niebezpieczeństwo, że zwolniony identyfikator procesu zostanie przydzielony przez system innemu procesowi. Dlatego też „osierocony" proces zostaje przejęty przez proces `init`, a jego PPID ustawiony na 1. Sytuacja taka jest jednak nienormalna (w niektórych systemach osierocone procesy nie mogą się poprawnie zakończyć i pozostają w systemie jako tzw. procesy-duchy).

Należy zapewnić, aby proces macierzysty poczekał na zakończenie swoich procesów potomnych i odebrał od nich kod zakończenia procesu. W tym celu proces macierzysty powinien wywołać funkcję `wait` tyle razy ile utworzył procesów potomnych. Funkcja ta ma następującą składnię:

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

int wait(int *stat_loc);
```

Funkcja zwraca identyfikator zakończonego procesu potomnego lub -1 jeżeli proces nie ma procesów potomnych.

W zmiennej wskazywanej przez `stat_loc` zapisywana jest liczba, która na starszym bajcie zawiera kod zakończenia procesu potomnego a na młodszym numer sygnału, który spowodował zakończenie działania procesu. Jeśli przedstawimy ją w postaci szesnastkowej jako XXYY, to XX oznacza kod zakończenia procesu potomnego, zaś YY numer sygnału, który spowodował zakończenie procesu potomnego lub 0, jeśli proces zakończył się samodzielnie.

Jedno wywołanie funkcji `wait` oczekuje tylko na zakończenie jednego procesu potomnego. Jeżeli nie wiemy ile mamy procesów potomnych, należy wykonywać w pętli funkcję `wait`, aż do momentu, kiedy zwróci wartość –1 (nie ma już procesów potomnych).

6. Sygnały

Sygnał to informacja dla procesu, że wystąpiło jakieś zdarzenie. Sygnały są wysyłane przez:
- jądro do procesu,
- proces do innego procesu.

Sygnały są zwykle asynchroniczne tzn. nie da się przewidzieć momentu ich pojawienia się. Proces może w dowolnym momencie otrzymać sygnał. Winien wówczas przerwać pracę i zareagować na otrzymany sygnał (wykonać odpowiednie operacje). Dlatego też sygnały nazywa się inaczej **przerwaniami programowymi**.

W systemie UNIX do każdego typu sygnału przypisane są określone czynności domyślne, które powinien wykonać proces po otrzymaniu danego sygnału. Mogą to być:
- zatrzymanie procesu
- zakończenie procesu
- zakończenie procesu z zapisaniem obrazu pamięci (utworzenie pliku core)
- ignorowanie

SIGNAL	ID	Action	Event	Command	Key
SIGHUP	1	Exit	Hangup	kill -HUP pid	
SIGINT	2	Exit	Interrupt	kill -INT pid	^C
SIGQUIT	3	Core	Quit	kill -QUIT pid	^\
SIGKILL	**9**	**Exit**	**Killed**	**kill -9 pid**	
SIGPIPE	13	Exit	Broken Pipe	kill -PIPE pid	
SIGTERM	15	Exit	Terminated	kill -TERM pid	
SIGSTOP	**23**	**Stop**	**Stopped (signal)**	**kill -STOP pid**	
SIGTSTP	24	Stop	Stopped (user)	kill -TSTP pid	^Z ^Y

Sygnał może być wysłany:

- z poziomu shell'a – poleceniem `kill`
  `kill -signal pid`
  np.: `kill -INT 2367`
  Listę sygnałów można wypisać poleceniem `kill -l`
- z poziomu programu – funkcją `kill`
  `int kill(int pid, int sig);`
- przez naciśnięcie klawisza (patrz tabela) – sygnał jest wysyłany do procesu działającego aktualnie na pierwszym planie
- przez jądro: błędy operacji, adresacji, arytmetyczne, pojawienie się wysokopriorytetowych danych w gnieździe itp.

Jeżeli wysyłając sygnał zamiast konkretnego numeru `pid` podamy wartość `0` to sygnał wysyłany do wszystkich procesów o takim samym identyfikatorze grupy procesów GPID jaki ma proces wysyłający sygnał.

Jeżeli zamiast konkretnego numeru `pid` podamy wartość `-1` i **nie jesteśmy administratorem** (nie root) to sygnał jest wysyłany do procesów o tym samym rzeczywistym id użytkownika (czyli wszystkich „naszych" procesów).

Jeżeli zamiast konkretnego numeru `pid` podamy wartość `-1` i **jesteśmy administratorem** (root) to sygnał będzie wysłany do wszystkich procesów niesystemowych.

Jeżeli zamiast konkretnego numery sygnału podamy wartość `0` to nie jest wysyłany żaden sygnał, ale możemy się dowiedzieć czy istnieje proces o danym numerze `pid`.

Wysłanie sygnału do wszystkich procesów o tym samym GPID jest przydatne między innymi wtedy, gdy chcemy natychmiast zakończyć działanie jakiejś aplikacji wieloprocesowej.

Każdy proces może zawierać swoje funkcje do obsługi sygnałów. Aby procedura obsługi sygnału została wywołana musi nastąpić **przechwycenie sygnału**. Można przechwycić każdy sygnał z wyjątkiem dwóch: **SIGKILL** i **SIGSTOP** (nie można ich przechwycić i nie można stworzyć własnej procedury do ich obsługi). Sygnały te nazywane są sygnałami niezawodnymi. Są one wyróżnione po to by administrator systemu miał możliwość zakończenia lub zatrzymania dowolnego procesu użytkownika.

Przechwycenie sygnału (zawsze określa się jaki numer sygnału ma być przechwycony):

- na stałe:
  `void (*sigset (int sig, void (*disp)(int)))(int);`
  gdzie zmienna `int sig` przekazuje numer sygnału, który ma być przechwycony zaś `void (*disp)(int)` jest wskaźnikiem do funkcji, która ma być wywołana w ramach reakcji na odebranie sygnału.
  w praktyce:
  - `sigset(SIGINT,proc);`
  - `sigset(SIGINT,SIG_IGN);` ignorowanie
  - `sigset(SIGINT,SIG_DFL);` wykonanie czynnosci domyślnej

- w niektórych systemach (orginalnie w UNIX oraz Linux kernel z libc4,5) istnieje funkcja do jednorazowego przechwycenia sygnału (pierwszy odebrany sygnał będzie przechwycony, kolejne już nie):
  `void (*signal (int sig, void (*disp)(int)))(int);`
  W systemach BSD oraz Linux kernel z glibc2 działanie tej wunkcji jest inne (zobacz manual)

- zablokowanie – sygnał będzie odebrany, ale zablokowany (nie zostanie wykonana żadna akcja). Po odblokowaniu sygnał będzie obsłużony.
  `int sighold(int sig);`

- odblokowanie
  `int sigrelse(int sig);`

- zignorowanie (tak jak `signal` z `SIG_IGN`) sygnał będzie zignorowany i stracony
  `int sigignore(int sig);`

7. Pytania kontrolne

1. Jakim poleceniem wyświetlić listę procesów?
2. Wymień identyfikatory określające właściciela procesu i uprawnienia procesu w systemie.
3. Jakie identyfikatory są związane z procesem?
4. Do czego może przydać się identyfikator GPID?
5. W jaki sposób można uruchomić proces?
6. Jak wyświetlić listę procesów uśpionych bądź działających w tle?
7. Do czego służy funkcja fork()?
8. Do czego służą funkcje z rodziny exec?
9. Jak spowodować, aby proces działał w systemie zawsze z prawami właściciela pliku?
10. Do czego służą bity „s” dla plików wykonywalnych?
11. Co to jest sygnał?
12. Jakie domyślne akcje są przypisane do różnych sygnałów?
13. Kto może wysłać sygnał?
14. Jak wysłać sygnał do wszystkich swoich procesów?
15. Co to są sygnały niezawodne?
16. Dlaczego proces macierzysty powinien zakończyć się później niż wszystkie jego procesy potomne?

8. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl`
Programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB07`

Ćwiczenie 1

- Skopiuj do swojego katalogu domowego program `testrun.c` i skompiluj.
- Uruchom go na pierwszym planie. Czy możesz teraz uruchamiać inne programy z wiersza poleceń?
- Zatrzymaj (uśpij) proces wysyłając sygnał TSTP (naciskając CTRL+Z). Czy teraz masz dostęp do wiersza poleceń?
- Zobacz listę procesów uśpionych i działających w tle.
- Uruchom drugi raz program `testrun`, tym razem w tle. Czy możesz teraz uruchamiać inne programy z wiersza poleceń?
- Zobacz listę procesów uśpionych i działających w tle.
- Przenieś proces działający w tle na pierwszy plan.
- Zakończ proces działający na pierwszym planie przez wysłanie sygnału INT (naciskając CTRL+C)
- Obudź uśpiony proces tak, aby działał na pierwszym planie.
- Zakończ działanie tego procesu

Ćwiczenie 2

Przejdź do katalogu `~rstankie/stud/LAB07.` Są tam między innymi pliki wykonywalne: `ids1`, `ids2`, `ids3`. Wszystkie one są kompilacją tego samego programu `ids.c`. Różnią się ustawieniem praw dostępu (bity „s"). Programy te wyświetlają na ekranie rzeczywiste i efektywne identyfikatory użytkownika i grupy, po czym kończą się po 15 sekundach bezczynności.

Sprawdź jakie są prawa dostępu do plików `ids1`, `ids2` i `ids3` oraz kto jest ich właścicielem (wartości numeryczne identyfikatorów właściciela uzyskasz dodając do `ls` opcję `-n`).

Uruchom `ids1` w tle. Wykonaj polecenie `ps -o uid,ruid,gid,rgid,fname -U` *`wlasny_login`*
Porównaj wartości identyfikatorów procesu ids1 wyświetlone przez niego samego i polecenie `ps`.

Uruchom `ids2` w tle. Wykonaj polecenie `ps -o uid,ruid,gid,rgid,fname -U` *`wlasny_login`*
Porównaj wartości identyfikatorów procesu ids2 wyświetlone przez niego samego i polecenie `ps`.

Uruchom `ids3` w tle. Wykonaj polecenie `ps -o uid,ruid,gid,rgid,fname -U` *`wlasny_login`*
Porównaj wartości identyfikatorów procesu ids3 wyświetlone przez niego samego i polecenie `ps`.

Zapamiętaj, jakie identyfikatory uległy zmianie przy ustawieniu bitów „s" dla użytkownika i/lub grupy.

Ćwiczenie 3

To ćwiczenie należy również wykonać w katalogu `~rstankie/stud/LAB07`

Spróbuj przeczytać pliki `test.1` i `test.2` za pomocą programu `cat`.
Dlaczego nie udał się odczytać drugiego pliku?

Spróbuj odczytać te pliki za pomocą programów `opcla` i `opclb` (oba są kompilacją programu `opcl.c`, różnią się jedynie prawami dostępu).

Dlaczego `opclb` pozwala przeczytać zawartość pliku `test.2` ?

Wróć do swojego katalogu domowego.

Ćwiczenie 4

Skopiuj do swojego katalogu domowego program `fk.c` i skompiluj. Przeanalizuj kod programu. Jakiego działania się spodziewasz?

Uruchom program `fk` w tle. Za pomocą polecenia `ps` Sprawdź ile powstało procesów o nazwie `fk` oraz jakie mają identyfikatory PID i PPID. Zwróć uwagę również na PID swojego shell'a. Poczekaj na zakończenie procesów (40s).

Powinny powstać 2 procesy. Macierzysty kończy działanie po 20 sekundach, potomny po 40 sekundach. Czy jest to sytuacja poprawna?

Uruchom ponownie program `fk` w tle. Wyświetl PID i PPID procesów zaraz po uruchomieniu, a następnie po zakończeniu procesu rodzica. Jaki jest teraz PPID procesu potomnego? Dlaczego?

Ćwiczenie 5
Skopiuj programy `exe.c` oraz `nowy.c` i skompiluj je (odpowiednio pod nazwami `exe` i `nowy` **!!ważne, żeby miały takie właśnie nazwy!!**). Uruchom program `exe` na pierwszym planie i zaobserwuj działanie.

Sprawdź, jaki został zwrócony kod zakończenia procesu (powinno być 7).

Uruchom ponownie `exe`, tym razem w tle i obserwuj PID, PPID i nazwę procesu, przed podmianą kodu i po. Co się zmieniło?

Przeanalizuj kody programów `exe.c` i `nowy.c`.
- Z których programów pochodzą poszczególne komunikaty?
- W programie `exe.c` są dwa wywołania funkcji `execl`. Czy drugi `exec` kiedykolwiek się wykona?
- Czy istnieje możliwość, że proces `exe` wywoła funkcję `return 5` ?

Skasuj skompilowany program `nowy`.
Uruchom ponownie program `exe`.
Jakie są teraz wyniki działania programu `exe`? Dlaczego?

Ćwiczenie 6
To ćwiczenie wygodnie będzie wykonywać mając uruchomione dwa terminale.

Skopiuj programy `csig1.c csig2.c csig3.c` oraz `dane.conf`

1. Uruchom program csig1. Spróbuj zatrzymać proces wysyłając sygnał TSTP (naciskając CTRL+Z lub wysyłając sygnał poleceniem `kill` z drugiego terminala). Jaki jest efekt?
   Wyślij do działającego procesu `csig1` sygnał HUP. Co się stało? Dlaczego?
   Sygnał HUP jest zwyczajowo używany do wysłania do procesu żądania odświeżenia jego parametrów konfiguracyjnych. Jeśli proces obsługuje ten sygnał, pobiera z odpowiedniego pliku dane i kontynuuje działanie.

2. Uruchom program `csig2`. Zaobserwuj reakcję na sygnały INT, TERM, TSTP, CONT. Przeanalizuj kod programu. Zwróć uwagę na obsługę sygnału TSTP. Proces zasypia po odebraniu tego sygnału, lecz uprzednio wysyła na ekran komunikat.

3. Uruchom program `csig3` na pierwszym planie. Wyślij do procesu sygnał INT po wyświetleniu komunikatu „Blokuje sygnał SIGINT”. Co się stanie po wyświetleniu komunikatu „Odblokowuje sygnał SIGINT”?

   Uruchom program jeszcze raz. Poczekaj na pojawienie się komunikatu „Odblokowuje sygnał SIGINT”. Wyślij sygnał INT.

   Przeanalizuj kod programu. Zwróć uwagę, że tym razem do obsługi sygnałów INT i TERM służy jedna funkcja. Rozróżnienie, który z tych dwóch sygnałów został odebrany jest możliwe.

Ćwiczenie 7
Należy napisać trzy programy: `nowy1.c`, `nowy2.c` oraz `fkexe.c`
Proces `fkexe` powinien po uruchomieniu utworzyć dwa procesy potomne, które następnie podmienią swoje kody na kody procesów `nowy1` i `nowy2`. Proces macierzysty powinien poczekać na zakończenie obu procesów potomnych. W momencie zakończenia któregoś procesu potomnego, proces macierzysty wyświetla na ekranie PID tego procesu oraz kod zakończenia procesu potomnego i numer sygnału, który spowodował zakończenie potomka. Jeśli nie ma już procesów potomnych, proces macierzysty również kończy swoje działanie.
Programy `nowy1.c` i `nowy2.c` powinny wyświetlić na ekranie jakiś komunikat i zawiesić swoje działanie na, odpowiednio, 20 i 30 sekund. Dodatkowo można w tych programach dopisać obsługę sygnałów INT i TERM, która spowoduje, że w przypadku zakończenia procesu sygnałem zostanie zwrócony kod zakończenia procesu różny od 0.

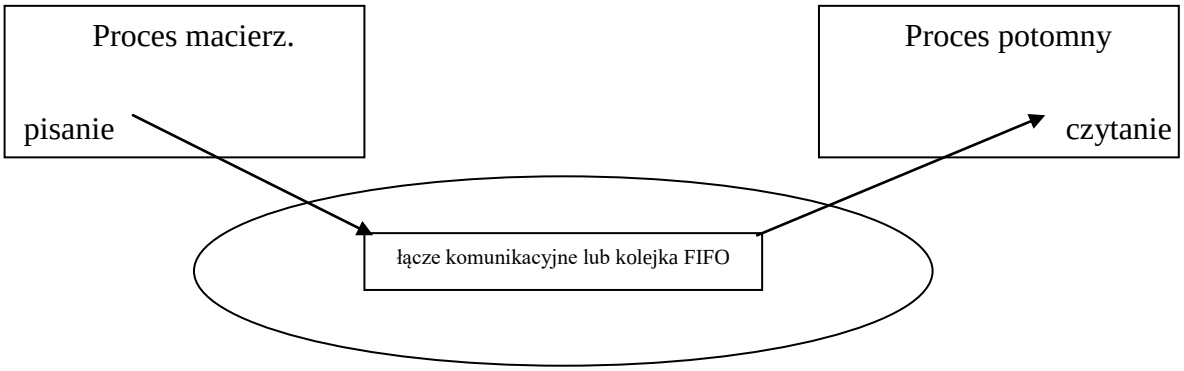
Laboratorium 8

- **Łącza komunikacyjne i kolejki FIFO**

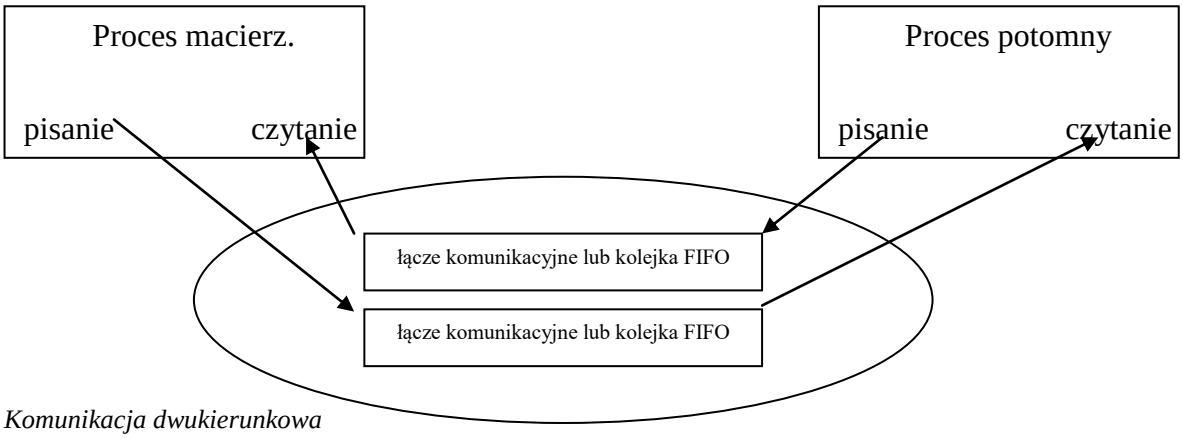
1. Wprowadzenie

Łącza komunikacyjne i kolejki FIFO umożliwiają przepływ danych w jednym kierunku. Korzysta się z nich tak samo jak z plików zwykłych odwołując się za pomocą deskryptora lub wskaźnika typu FILE *.

Proces może otworzyć jedno łącze lub kolejkę FIFO zarówno do pisania jak i czytania.



Komunikacja jednokierunkowa



Komunikacja dwukierunkowa

2. Kolejki FIFO - named pipe

Kolejka FIFO jest tworzona na dysku jako plik specjalny typu kolejka fifo. Dzięki temu, że kolejki FIFO mają nazwę, mogą być stosowane do komunikacji między dowolnymi procesami (procesy nie muszą być spokrewnione).

```
prw-r--r--   1 rstankie telecom        0 May  9 16:13 plik_fifo
```

Plik taki otwiera się jak każdy inny plik funkcją `open` lub `fopen`. Podobnie należy go później zamknąć funkcją `close` lub `fclose`. Jeden z procesów otwiera kolejkę tylko do czytania a drugi tylko do pisania. Zapis i odczyt jest wykonywany podobnie jak w przypadku pliku zwykłego.

Do zadziałania łącza jest konieczne przyłączenie procesów z obydwu stron. Jeżeli tylko jeden proces otworzy łącze np. do czytania to zostanie zawieszony w funkcji otwierającej do momentu kiedy inny proces otworzy łącze do pisania. (Jeden proces może też otworzyć łącze równocześnie do czytania i do pisania).

Jeżeli z jakichś powodów proces odbierający dane zakończy się, to do procesu piszącego wysyłany jest sygnał BrokenPipe (SIGPIPE). Domyślnie proces piszący również się zakończy.
Jeśli jako pierwszy zakończy się proces piszący, proces czytający będzie domyślnie działał nadal, co najwyżej zatrzyma się na funkcji odczytu (w zależności od tego jak został napisany) do momentu pojawiania się tam jakichś danych.

Jeżeli plik „pipe" nie istnieje na dysku to można go utworzyć:

- programowo:
```
int mknode(char *path, int mode, int dev); z flagą S_IFIFO
int mkfifo(char *path, int mode);
```
- z linii komend:
```
mknod nazwa p
mkfifo nazwa
```

Usunięcie pliku:

- programowo:
```
int unlink(char *path);
```
- z linii komend
```
rm nazwa
```

3. Łącza komunikacyjne - unnamed pipe

Łącza komunikacyjne w odróżnieniu od kolejek FIFO nie mają nazwy. Mogą być, więc używane tylko do komunikacji między procesami spokrewnionymi.

Łącze komunikacyjne jest tworzone przez funkcję:
```
int pipe(int *descr);
```

Generowana jest para deskryptorów: otwarty do czytania `descr[0]` i otwarty do pisania `descr[1]`. Jeżeli dokonujemy rozwidlenia procesów to zarówno macierzysty jak i potomny dziedziczą parę deskryptorów. Wówczas na samym początku każdy z nich powinien zamknąć jeden z deskryptorów.

Z tego mechanizmu korzysta interpreter poleceń, pozwalając tworzyć tzw. potoki. Wyniki wykonania jednego procesu są wówczas przekazywane przez łącze komunikacyjne na wejście innego procesu itd.

Przykład:

```
ls -l
ls -l | grep rw-
ls -l | grep rw- | sort +4
```

Do realizacji potoków interpreter poleceń korzysta z funkcji `int dup2(int fildes, int fildes2);` która powoduje, że deskryptor pliku `fildes2` będzie się odwoływał do tego samego pliku co `fildes`. Przed dokonaniem podmiany plik `fildes2` jest zamykany.

W momencie uruchamiania każdy proces ma otwarte trzy deskryptory: 0, 1 i 2 odpowiadające stdin, stdout, stderr. Za pomocą funkcji `dup2` można podmienić je np. na deskryptory otrzymane przy tworzeniu łącza komunikacyjnego:

Przykład:

```
int descr[2];   /*tablica deskryptorow */
pipe(descr);
dup2(descr[1],1);
printf("aaa\n");
```

W powyższym przykładzie funkcja printf wpisze ciąg znaków "aaa" do utworzonego łącza komunikacyjnego (oryginalne stdout zostało podmienione na to łącze).

4. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl`
Programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB08`

Ćwiczenie 1:

Skopiuj i skompiluj programy: `pisz.c`, `czyt.c`. Program `pisz` służy do wprowadzania danych do kolejki fifo (named pipe), zaś program `czyt` do odbierania danych. Oba programy można zakończyć jedynie przez wysłanie do nich sygnału. Programy mają ustawione przechwytywanie sygnałów `INT` i `TERM`. Dodatkowo, program `pisz` przechwytuje sygnał `PIPE`.

Przeanalizuj kody obu programów.
- Na jakiej podstawie proces czytelnika orientuje się, że przestał istnieć proces pisarza?
- Na jakiej podstawie proces pisarza orientuje się, że przestał istnieć proces czytelnika?

Przed rozpoczęciem wykonywania poniższych ćwiczeń utwórz plik specjalny typu kolejka fifo o nazwie `plik_fifo` w tym samym katalogu, w którym znajdują się programy `czyt` i `pisz`.

a)
- W jednym oknie uruchom program `czyt`, a w drugim `pisz`. Zwróć uwagę, w którym momencie dochodzi do faktycznego otwarcia pliku `plik_fifo`, przez oba procesy. W którym miejscu wisi pierwszy z uruchamianych procesów?
- Prześlij jakieś dane przez kolejkę fifo.
- Za pomocą sygnału `INT` lub `TERM` zakończ proces `czyt`. Czy proces `pisz` działa nadal?

b)
- Uruchom ponownie programy `czyt` i `pisz`. Prześlij jakieś dane.
- Za pomocą sygnału `INT` lub `TERM` zakończ proces `pisz`. Czy proces `czyt` działa nadal?
- Pozwól procesowi `czyt` kontynuować działanie.
- Uruchom ponownie proces `pisz`. Czy dane są nadal przekazywane do procesu `czyt`?
- W trzecim oknie uruchom drugi proces `pisz`. Czy proces `czyt` odbiera dane z obu procesów?
- Zakończ działanie wszystkich procesów

Ćwiczenie 2:

Skopiuj i skompiluj program `upipe.c`. Przeanalizuj kod.
Proces główny tworzy łącze komunikacyjne (unnamed pipe), po czym tworzy dwa procesy potomne: czytelnika i pisarza. Dalsze działanie programu jest zbliżone do działania programów z ćwiczenia 1. Sprawdź, co się stanie, gdy zakończymy sygnałem proces czytelnika lub proces pisarza.

Czy do łącza komunikacyjnego można dołączyć trzeci proces (pisarza) tak jak w przedostatnim punkcie ćwiczenia 1?

Ćwiczenie 3:

Napisz program `upipe_ex.c`, który uruchomi jako osobne procesy polecenia „`cat /etc/passwd`” oraz „`sort`” oraz połączy je za pomocą łącza komunikacyjnego tak, aby uzyskać wynik identyczny z wynikami wykonania następującego wywołania z wiersza poleceń:

```
cat /etc/passwd | sort
```

Zalecane użycie funkcji `dup2`.

Laboratorium 9

- **Gniazda, komunikacja sieciowa**

1. Model sieci

Dla opisania rodziny protokołów Internetu często stosuje się uproszczony model czterowarstwowy. Model ten oraz jego relację do modelu siedmiowarstwowego OSI/ISO przedstawiono na rys.1.

Rys 1. Model czterowarstwowy rodziny protokołów Internetu oraz model OSI-ISO

W warstwie dostępu do sieci znajduje się interfejs sprzętowy. Konkretnymi protokołami tej warstwy nie będziemy się zajmować. W warstwie sieciowej można wymienić takie protokoły jak IP, ICMP, ARP, RARP. W warstwie transportowej można wymienić wiele protokołów. W naszych rozważaniach zajmiemy się jedynie protokołami TCP i UDP. W warstwie aplikacji znajdują się działające procesy.

TCP jest protokołem połączeniowym. Przed przesłaniem danych użytkownika, konieczne jest zestawienie połączenia, potem następuje faza przesyłania danych, następnie połączenie jest zamykane. Protokół TCP jest wyposażony w mechanizmy zapobiegające powstawaniu przeciążeniom w sieci, między innymi poprzez dostosowanie szybkości przesyłania danych do stanu sieci. TCP dysponuje również systemem potwierdzeń, co pozwala kontrolować czy wysłane pakiety dotarły do odbiorcy.

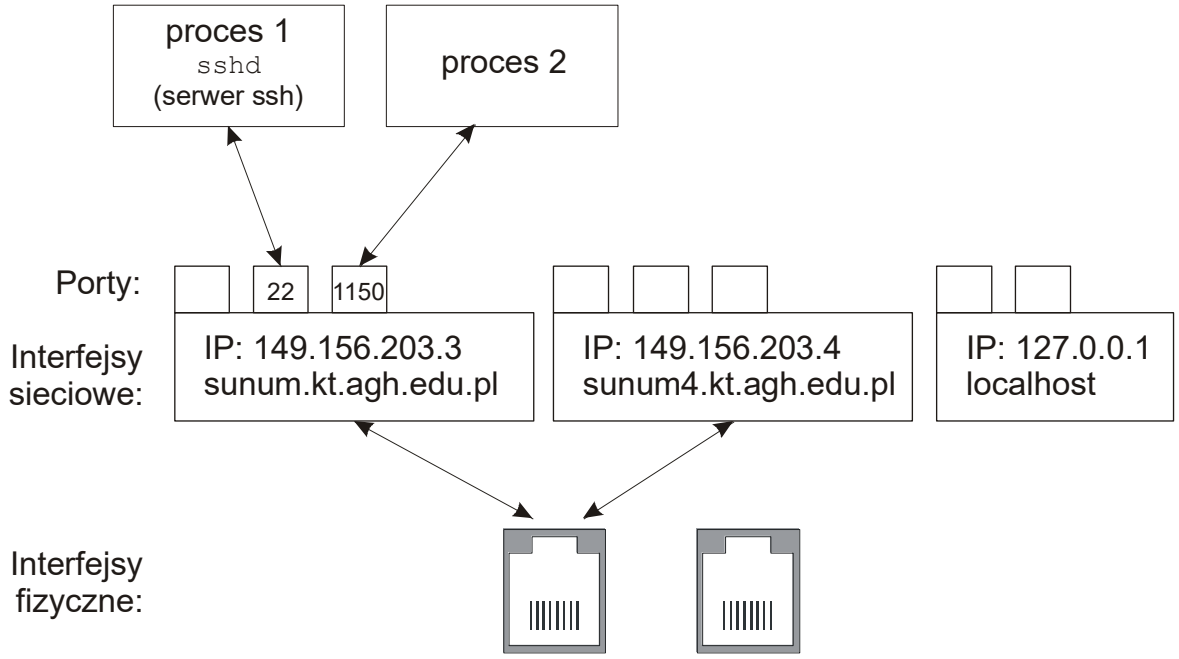
UDP jest protokołem bezpołączeniowym, nie posiadających mechanizmów sprawdzania poprawności dostarczenia danych do miejsca przeznaczenia. Pakiety są wysyłane na wskazany adres, a protokół nie dba o to czy dotarły. Może wysyłać pakiety z dowolną szybkością (ograniczoną szybkością interfejsu i łącza), UDP nie adaptuje prędkości wysyłania pakietów do stanu sieci.

2. Interfejsy sieciowe, adresy IP, porty

W każdym komputerze może znajdować się jeden lub więcej interfejsów sprzętowych. Na każdym interfejsie sprzętowym może znajdować się wiele interfejsów sieciowych (np. pojedyncza karta sieciowa w komputerze może mieć wiele adresów IP). W każdym komputerze istnieje interfejs sieciowy loopback o adresie 127.0.0.1 (localhost). Nie jest on związany z żadnym interfejsem fizycznym i jest dostępny tylko z tego komputera.

Adres IP nie wystarcza jednak do precyzyjnego określenia nadawcy/odbiorcy pakietów. Konieczne jest określenie, kto (jaki proces) powinien obsłużyć pakiety przychodzące na dany adres sieciowy. Służą do tego porty. Trójka adres IP – numer portu – protokół jest jednoznacznym adresem procesu (aplikacji) w całej sieci Internet. Na każdym interfejsie sieciowym może znajdować się wiele portów. W danej chwili na jednym porcie może nasłuchiwać tylko jedna aplikacja (proces). Schematycznie przedstawiono to na rys. 2. Z drugiej strony, jeden proces może nasłuchiwać na wybranym porcie jednocześnie na wielu interfejsach sieciowych.

Wiele serwisów ma przydzielone i ogólnie znane porty np. http – 80, ftp – 21, pop3 – 110, ssh – 22 itp. Dla protokołów UDP i TCP numery portów od 1 do 1023 są tzw. ogólnie znanymi portami. Standardowo są do nich przypisane nazwy protokołów warstwy aplikacji. Zwykle nie jest to wprost nazwa konkretnej aplikacji słuchającej na tym porcie. Przykładowo aplikacją realizującą funkcje serwera www słuchającą na porcie 80 może być `apache`, `AOL web serwer`, `Microsoft IIS` itp. Wiadomo jedynie, że na porcie 80 można się komunikować używając protokołu http natomiast nie jest oczywiste, jaka aplikacja to realizuje. Przypisanie nazw protokołów warstwy aplikacji do poszczególnych portów można znaleźć w pliku `/etc/services`. Użytkownicy niestandardowych programów muszą sami dbać o to, żeby numer portu serwera był znany klientom.

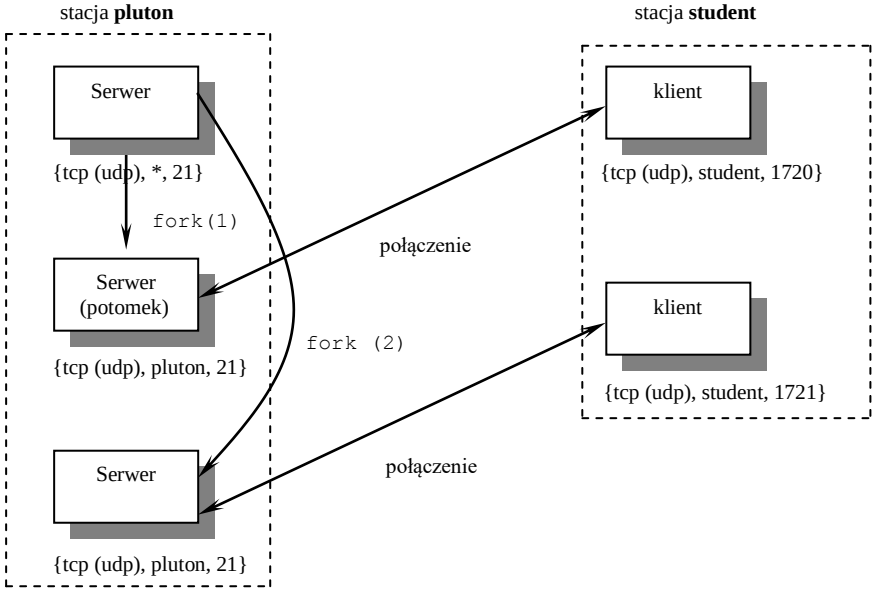


Rys.2 Interfejsy fizyczne, sieciowe i porty

3. Model klient-serwer

Standardowym modelem komunikacji w sieci Internet jest model klient-serwer. Proces serwera zostaje uruchomiony na pewnym komputerze, po czym przechodzi w stan oczekiwania na żądania obsługi od procesów klientów. Serwer oczekuje na dane klienta na określonym interfejsie sieciowym i numerze portu i z użyciem określonego protokołu transportowego. Można wyróżnić dwa typy serwerów: iteracyjne oraz współbieżne. Serwer iteracyjny obsługuje w danej chwili żądanie tylko jednego klienta (zakłada się, że czas potrzebny na obsługę jest znany i stosunkowo krótki). Serwer współbieżny może obsługiwać wiele procesów klientów jednocześnie, a czas obsługi nie jest z góry znany i może być długi. Serwer współbieżny tworzy osobne procesy dedykowane do obsługi poszczególnych klientów, co pokazuje rysunek 3.

Ze względu na pewne cechy wspólne protokoły są zgrupowane w rodziny protokołów. Można wyróżnić następujące rodziny protokołów: protokoły wewnętrzne UNIX-a, protokoły Internetu, protokoły Xerox-a NS, protokoły warstwy kanałowej IMP. My będziemy zajmować się jedynie protokołami Internetu.



Rys. 3 Schemat działania serwera współbieżnego.

4. Asocjacje

Komunikujące się procesy są w pełni określone przez asocjację. Asocjacja jest zbiorem pięcioelementowym postaci:
`{protokół, adres lokalny, proces lokalny, adres obcy, proces obcy}`.
Poprawna asocjacja dla rodziny protokołów Internetu może mieć postać (adresy lokalny i obcy będą adresami IP komputerów, na których działają komunikujące się procesy, zaś proces lokalny i proces obcy będą to numery portów na tych komputerach):
`{tcp, 149.156.203.195, 1500, 149.156.114.3, 21}`.

Należy zdefiniować jeszcze półasocjację. Ma ona postać: `{protokół, adres lokalny, port lokalny}` lub `{protokół, adres obcy, port obcy}`
Półasocjacja nazywana jest też **gniazdem** (ang. *socket*). Gniazdo jest jednym z końcowych punktów komunikacji. Z gniazd korzysta się w podobny sposób jak ze zwykłych plików. Z punktu widzenia systemu operacyjnego gniazdo jest plikiem specjalnym identyfikowanym przez deskryptor. Gniazda są dwukierunkowe (można zarówno do nich pisać, jak i z nich czytać).

Półasocjacja powstaje w dwóch etapach. Najpierw tworzy się gniazdo bez dowiązania określając jedynie rodzinę adresów typ gniazda oraz protokół. Na tym etapie uzyskuje się deskryptor gniazda. Do utworzenia pełnej asocjacji konieczne jest określenie pozostałych jej elementów. Proces ten nazywa się dowiązaniem (ang. *bind*).

5. Oglądanie gniazd i zestawionych połączeń za pomocą netstat

Polecenie `netstat` wyświetla listę istniejących w systemie gniazd oraz stany w jakich się znajdują. Nazwy poszczególnych stanów gniazda mają różne znaczenie dla protokołów TCP i UDP. Przykładowe wyniki wykonania polecenia `netstat` dla protokołu UDP zaprezentowano poniżej:

```
UDP: IPv4
   Local Address         Remote Address     State
-------------------- -------------------- -------
      *.*                                  Unbound
      *.32771                              Idle
      *.time                               Idle
pluton.6971                                 Idle
192.34.145.12.6970                         Idle
localhost.48467      localhost.48467       Connected
```

Gniazdo o adresie lokalnym `*.*` jest gniazdem, dla którego określono jedynie protokół, zaś nie określono jeszcze pozostałych elementów półasocjacji. Jest ono w stanie `Unbound` czyli w jest niedowiązane.
Oznaczenie `*.32771` informuje nas, że istnieje jakiś proces korzystający z tego gniazda i odbiera pakiety, które mają w adresie docelowym numer portu 32771 oraz dowolny adres IP spośród adresów interfejsów sieciowych działających na tym komputerze. Gniazdo jest w stanie oczekiwania (`Idle`), czyli może odbierać przychodzące datagramy.

Wpis `*.time` jest analogiczny, tylko że numer portu został zastąpiony nazwą protokołu warstwy aplikacji (na podstawie wpisu w pliku `/etc/services`). Podobnie adres interfejsu sieciowego może być wyrażony numerycznie bądź w postaci nazwy domenowej. Powiązanie nazw domenowych z adresami IP jest przechowywane przez serwery DNS (ang. *Domain Name Server*). Jeżeli jest określony adres interfejsu sieciowego, oznacza to, że dana aplikacja odbiera pakiety kierowane wyłącznie na ten interfejs sieciowy. Symbol * (na lewo od kropki) oznacza zawsze, że aplikacja odbiera dane na wskazanym porcie, ale na dowolnym interfejsie sieciowym.

Ostatni wpis może być nieco mylący. Należy pamiętać, że UDP jest protokołem bezpołączeniowy, więc nigdy w sensie dosłownym nie jest ustanawiane połączenie między klientem i serwerem. Stan `Connected` oznacza tu jedynie, że datagramy wysyłane z gniazda określonego jako `local address` będą wysyłane wyłącznie na adres określony jako `remote address`. Jest to niejako ograniczenie funkcjonalności gniazda (na stałe jest ustalony adres docelowy), ale z drugiej strony, upraszcza to procedury wysyłania danych (o czym będzie też mowa dalej).

Poniżej przedstawiono przykładowe wyniki wykonania polecenia `netstat` dla protokołu TCP:

```
TCP: IPv4
  Local Address      Remote Address    Swind Send-Q Rwind Recv-Q  State
----------------- -------------------- ----- ------ ----- ------ -------
      *.*              *.*                0      0  49152      0 IDLE
      *.64174          *.*                0      0  49152      0 BOUND
      *.80             *.*                0      0  49152      0 LISTEN
pluton.ssh        rafi.kt.agh.edu.pl.3234  65039  47 48480      0 ESTABLISHED
pluton.80         208.22.177.10.28087      65011  0  49298      0 TIME_WAIT
pluton.36504      145.230.2.6.ssh          0      0  49640      0 SYN_SENT
```

Pole `remote address` jest wypełnione tylko wtedy, gdy proces jest w jednym z etapów nawiązywania bądź rozłączania połączenia TCP lub połączenie jest zestawione. Wówczas znana jest pełna asocjacja. W pozostałych przypadkach pole to ma wartość `*.*`, co oznacza, że druga połowa asocjacji jest nieokreślona.

Stan `IDLE` oznacza, że gniazdo TCP jest utworzone, ale określono jedynie protokół, zaś nie określono jeszcze pozostałych elementów półasocjacji (odpowiednik stanu `Unbound` dla UDP). Stan `BOUND` oznacza, że znane są już wszystkie elementy półasocjcacji, ale proces, który dowiązał gniazdo jeszcze nie jest w stanie obsługiwać przychodzących połączeń. Stan `LISTEN` oznacza, że gniazdo jest w stanie nasłuchiwania, czyli jest gotowe do obsługi przychodzących połączeń. Stan `ESTABLISHED` oznacza, że połączenie jest ustanowione. Gniazdo może znajdować się również w jednym z etapów nawiązywania połączenia (np. `SYN_SENT`), bądź rozłączania połączenia (np. `TIME_WAIT`). Szczegółowy opis stanów połączeń TCP można znaleźć w RFC 793.

Gniazdo w stanie `BOUND` lub `LISTEN` może raportować lokalny adres IP jako * , co oznacza dowiązanie do każdego z istniejących interfejsów sieciowych. Jeżeli połączenie jest w trakcie zestawiania, jest ustanowione lub jest rozłączane to interfejs sieciowy jest zawsze ściśle określony (proces komunikuje się przez jeden interfejs sieciowy, a nie przez kilka naraz). Pełna asocjacja ma ściśle określone adresy komunikujących się procesów.

Uwaga: W systemie działającym obecnie na plutonie `netstat` nie pokazuje gniazd w stanie `IDLE` i `BOUND`.

6. Funkcje do operacji na gniazdach

Opis podanych funkcji będzie koncentrował się na rodzinie protokołów Internetu, w szczególności protokołów TCP i UDP. W praktyce, funkcje te mają zastosowanie uniwersalne do różnych typów gniazd.

```
<sys/types.h>
<sys/socket.h>
int socket (int family, int type, int protocol);
```

Jest to funkcja systemowa, za pomocą której proces określa typ gniazda i rodzinę protokołu, z którego gniazdo będzie korzystać. Funkcja zwraca deskryptor gniazda.
Jako argument family (rodzina) można wybrać między innymi jedną ze stałych:

`AF_UNIX` – protokoły Unixa
`AF_INET` – protokoły Internetu
`AF_NS` – protokoły Xeroxa NS

Jako argument `type`, określający typ gniazda, można użyć jednej z poniższych stałych:

`SOCK_STREAM` – gniazdo strumieniowe
`SOCK_DGRAM` – gniazdo datagramowe
`SOCK_RAW` – gniazdo surowe
`SOCK_SEQPACKET` – gniazdo pakietów uporządkowanych
`SPCK_RDM` – gniazdo komunikatów niezawodnie doręczanych

Trzeci argument określa protokół. Mogą tam wystąpić między innymi wartości:

`IPPROTO_UDP` – protokół UDP
`IPPROTO_TCP` – protokół TCP
`IPPROTO_ICMP` – protokół ICMP

Stałe te są zdefiniowane w nagłówku `<netinet/in.h>`.

Nie wszystkie kombinacje rodzin protokołów, rodzajów gniazd i nazw protokołów są poprawne. Poprawne kombinacje dla rodziny protokołów Internetu zestawiono w poniższej tabeli.

rodzina	typ	protokół	protokół rzeczywisty
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	(surowy)

```
<sys/types.h>
<sys/socket.h>
int bind (int sockfd, struct sockaddr *myaddr, int addrlen);
```

Funkcja systemowa `bind` dowiązuje do gniazda adres i port lokalny. Pierwszy argument określa deskryptor gniazda. Drugi argument jest wskaźnikiem do struktury zawierającej adres i port, który ma być dowiązany. Trzeci argument określa rozmiar struktury zawierające adres. Jest on konieczny, ze względu na uniwersalność funkcji; inne rodziny protokołów używają innej struktury do przechowywania adresów.

Proces serwera (TCP bądź UDP) musi wywołać tę funkcję. Za jej pośrednictwem informuje system o tym, na którym porcie (ewentualnie również konkretnym interfejsie sieciowym) będzie oczekiwał na dane. Dzięki temu system wie, że pakiety przesyłane pod ten adres mają trafiać do tego serwera.

Klient TCP nie musi wywoływać tej funkcji. Wystarczy, że wywoła (omówioną dalej) funkcję connect. Wywołanie to powoduje wysłanie żądania nawiązania połączenia z serwerem, a system klienta samodzielnie przydzieli mu adres i port lokalny. Taki adres lokalny jest ważny przez cały czas trwania połączenia. Jednakże klient może wywołać funkcję `bind`, aby ustalić na stałe adres i port, którego będzie używał do komunikacji.

W przypadku transmisji bezpołączeniowej klient musi mieć na stałe ustalony adres lokalny. Gdyby adres ten się zmieniał (był różny dla poszczególnych wysyłanych do serwera datagramów) serwer nie miałby możliwości wysłania odpowiedzi. Klient UDP musi mieć jednoznaczny adres, aby serwer mógł wysłać do niego odpowiedzi.

```
<sys/types.h>
<sys/socket.h>
int connect (int sockfd, struct sockaddr *servaddr, int addrlen);
```

W przypadku protokołu TCP funkcja `connect` służy do nawiązania połączenia z serwerem przez klienta. Pierwszym argumentem jest deskryptor gniazda, drugim wskaźnik do struktury, w której umieszczony powinien być adres serwera. Trzeci argument określa rozmiar struktury zawierającej adres.

Klient UDP również może użyć funkcji `connect`. Należy jednak pamiętać, że w tym wypadku **nie jest to ustanawianie połączenia**. W takiej sytuacji, proces klienta niejako informuje system, że wszystkie wysyłane datagramy będą kierowane na przekazany w funkcji `connect` adres serwera. W takiej sytuacji proces klienta może posługiwać się prostszymi funkcjami do zapisu/odczytu danych z gniazda, takimi jak `read, write, recv i send`. W przeciwnym razie (jeśli nie użyje funkcji `connect`) musi do wysłania danych użyć funkcji `sendto`, zaś do odbioru `recvfrom`, które przekazują adres serwera.

```
<sys/types.h>
<sys/socket.h>
int listen (int sockfd, int backlog);
```

Funkcja `listen` musi być wywołana przez serwer połączeniowy (TCP) w celu przejścia w stan gotowości na przyjmowanie połączeń. Pierwszy argument to deskryptor gniazda. Drugi argument określa maksymalną liczbę żądań nawiązania połączenia, które mogą oczekiwać na obsługę. Zwykle przyjmuje się wartość 5. Serwer jest w stanie nawiązywać w danej chwili jedno połączenie. W tym czasie mogą przyjść inne żądania, które są wówczas ustawiane w kolejce. Jeżeli połączenie zostanie nawiązane, proces serwera powinien przekazać obsługę tego połączenia procesowi potomnemu, a sam zająć się obsługą następnego żądania nawiązania połączenia.

Podsumowując, serwer współbieżny TCP może obsługiwać wiele zestawionych połączeń, zaś nawiązywać może w danej chwili tylko jedno. Pozostałe czekaja w kolejce, a długość kolejki jest określona przez drugi argument funkcji `listen`.

```
<sys/types.h>
<sys/socket.h>
int accept (int sockfd, struct sockaddr *peer, int addrlen);
```

Proces serwera, po wejściu w stan oczekiwania wywołuje funkcję accept. Jest to funkcja blokująca, co oznacza, że proces serwera jest zatrzymany do momentu pojawienia się jakiegoś żądania nawiązania połączenia. Jeżeli takie żądanie się pojawi, funkcja `accept` tworzy nowe gniazdo dedykowane dla tego połączenia (zwraca jego deskryptor) i przeprowadza procedury ustanawiające połączenie TCP. Po ustanowieniu połączenia proces opuści tę funkcję. Powinien wówczas utworzyć proces potomny, który zajmie się obsługą połączenia. Do potomka jest przekazywany deskryptor nowego gniazda (potomek zamyka deskryptor gniazda oryginalnego). Proces serwera zamyka deskryptor nowego gniazda, po czym w pętli wywołuje po raz kolejny funkcję `accept`, aby móc przyjąć kolejne żądanie nawiązania połączenia.

Pierwszy argument funkcji `accept` to deskryptor gniazda (oryginalnego), na którym serwer oczekuje na połączenia. Drugi to wskaźnik do struktury, w której zostanie zapisany adres klienta, który nawiązał połączenie.

```
<sys/types.h>
<sys/socket.h>

int send ( int sockfd, char *buff, int nbytes, int flags );

int sendto ( int sockfd, char *buff, int nbytes, int flags, struct sockaddr
*to, int addrlen );

int recv ( int sockfd, char *buff, int nbytes, int flags );

int recvfrom ( int sockfd, char *buff, int nbytes, int flags, struct
sockaddr *from, int addrlen );
```

Funkcje systemowe send, sendto, recv i recvfrom są podobne do zwykłych funkcji systemowych `read` oraz `write`. Pierwsze trzy argumenty określają odpowiednio deskryptor gniazda, adres bufora i liczbę zapisanych/odczytanych bajtów. Argument flags może mieć wartość zero lub jedną z poniższych wartości (przykładowe dwie możliwości):

`MSG_OOB` – wyślij lub odbierz dane wysokopriorytetowe

`MSG_DONTROUTE` – obejdź wybieranie trasy (`send` lub `sendto`)

Argument `to` w funkcji `sendto` określa właściwy dla danego protokołu adres, pod który mają być przesłane dane. Argument `addrlen` określa długość adresu. Funkcja `recvfrom` przekazuje poprzez argument `from` adres, spod którego zostały odebrane dane.

Funkcja systemowa służąca do zamykania gniazda:

```
int close (int fd);
```

7. Struktury do przechowywania adresów

Dla rodziny protokołów Internetowych zdefiniowano następujące struktury do przechowywania adresów (<netinet/in.h>):

```
struct in_addr {
      u_long s_addr;
};
```

Element `s_addr` przechowuje adres IP w postaci liczby 32-bitowej, zachowując sieciową kolejność bajtów.

```
struct sockaddr_in {
      short             sin_family;
      u_short           sin_port;
      struct in_addr    sin_addr;
      char              sin_zero[8];
}
```

Element `sin_family` ma zawsze wartość `AF_INET`. Element `sin_port` przechowuje numer portu. Element `sin_addr` przechowuje adres IP w postaci opisanej wyżej struktury. Ostatni element jest nieużywany.

Kolejność bajtów ...

Różne systemy różnią się między sobą sposobem przechowywania liczb większych niż 1 bajt. Jeżeli bajt o niższym adresie w pamięci jest bajtem mniej znaczącym to mamy do czynienia ze sposobem „mniejsze niżej" (little endian). W przeciwnym wypadku mówimy o sposobie „mniejsze wyżej" (big endian). W sieci Internet przyjęto sposób przechowywania liczb „mniejsze wyżej". Jednakże, jak już wspomniano, komputer przyłączony do sieci może mieć inny sposób przechowywania liczb. Wyróżnia się sieciową kolejność bajtów (network) oraz kolejność bajtów stacji (host). Dlatego, aby uniknąć niejednoznaczności, w każdym programie zaleca się zawsze dokonywać konwersji liczb z sieciowej kolejności bajtów na kolejność bajtów stacji i odwrotnie. Operacje te są wspierane przez następujące funkcje:

```
<sys/types.h>
<netinet/in.h>

u_long  htonl(u_long hostlong);      konwersja z postaci stacji na postać sieciową (liczba typu long)
u_short htons(u_short hostshort);    konwersja z postaci stacji na postać sieciową (liczba typu short)
u_long  ntohl(u_long netlong);       konwersja z postaci sieciowej na postać stacji (liczba typu long)
u_short ntohs(u_short netshort);     konwersja z postaci sieciowej na postać stacji (liczba typu short)
```

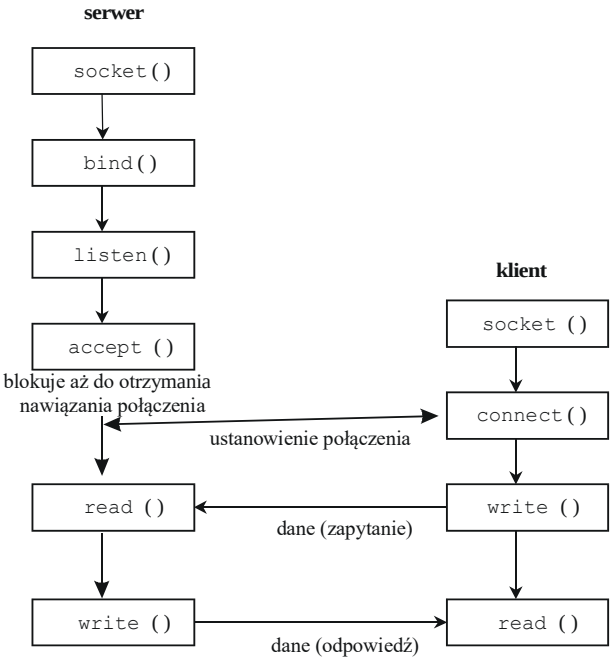
Funkcje do konwersji adresów

Wygodnym sposobem przedstawiania adresu IP jest postać czterech liczb rozdzielonych kropkami. W rzeczywistości jednak, adres jest liczbą typu long. Do odpowiednich konwersji stosuje się tu dwie funkcje:

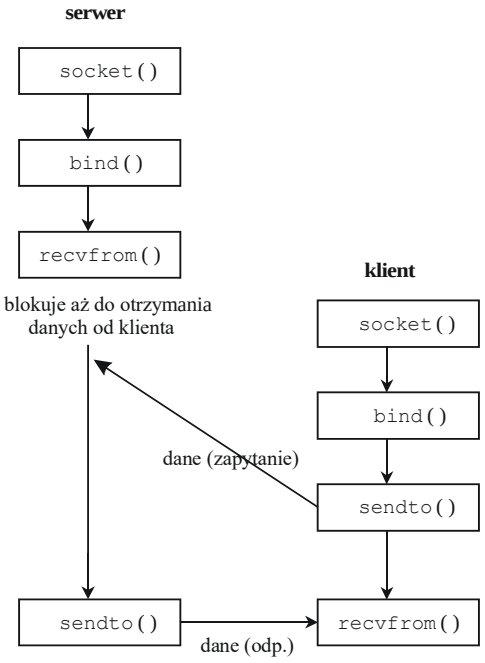
```
<sys/socket.h>
<netinet/in.h>
<arpa/inet.h>

unsigned long     inet_addr(char *ptr);
char  *inet_ntoa(struct in_addr inaddr);
```

8. Schemat komunikacji

Na poniższym rysunku przedstawiono schematy komunikacji dla aplikacji klient-serwer posługujących się protokołami TCP i UDP.



Użycie funkcji systemowych w protokole połączeniowym.



Użycie funkcji systemowych w protokole bezpołączeniowym.

9. Pytania kontrolne

1. Do czego odnosi się adres IP 127.0.0.1?
2. Czym różnią się protokoły TCP i UDP?
3. Podaj różnice między modelem siedmiowarstwowym OSI/ISO i modelem czterowarstwowym rodziny protokołów Internetu
4. Co to są „ogólnie znane numery portów”? Podaj przykłady.
5. Co to jest interfejs fizyczny?
6. Co to jest interfejs sieciowy?
7. Co to jest asocjacja i półasocjacja?
8. W jaki sposób można jednoznacznie określić adres procesu (aplikacji) w Internecie?
9. Opisz model klient-serwer. Jak działają serwery współbieżne a jak iteracyjne?
10. Zinterpretuj wynik działania polecenia netstat dla protokołu UDP:

```
Local Address         Remote Address     State
-------------------- -------------------- -------
192.34.145.12.6970                          Idle
```

11. Zinterpretuj wynik działania polecenia netstat dla protokołu TCP:

```
  Local Address      Remote Address    Swind Send-Q Rwind Recv-Q  State
---------------- -------------------- ----- ------ ----- ------ -------
      *.80             *.*                0      0  49152      0 LISTEN
pluton.ssh            rafi.kt.agh.edu.pl.3234   65039   47  48480         0
ESTABLISHED
```

12. Wymień rodzaje gniazd
13. Gdzie można znaleźć przypisanie numerów portów do nazw protokołów?
14. Gdzie można znaleźć adres IP dla znanego adresu domenowego?
15. Ile procesów może jednocześnie nasłuchiwać na porcie TCP/80 na interfejsie sieciowym 127.0.0.1?

10. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl` Programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB09`

Ćwiczenie 1:

Uruchom komputer PC z systemem Ubuntu. Z konta administratora lub wykorzystując polecenie `sudo` wykonaj następujące polecenia:
`iptables -F`
oraz
`service ssh start` (jeśli serwer ssh nie działa)
Pierwsze polecenie odblokowuje zabezpieczenia sieciowe w serwerze, drugie uruchamia serwer SSH.
Zaloguj się na serwer pluton.kt.agh.edu.pl:
`ssh` *`twój_login_na_plutonie`*`@pluton.kt.agh.edu.pl`

Wykonaj polecenie `netstat –a | less`.
Przeanalizuj wyniki. Spróbuj je zinterpretować. Jeżeli jakaś informacja jest niejasna, poszukaj wyjaśnienia w materiałach. Jeśli wszystko zawiedzie – spytaj prowadzącego ☺

Przydatna jest opcja `–n`, która powoduje, że adresy IP nie są zamieniane na nazwy domenowe, zaś numery portów nie są zamieniane na nazwy aplikacji. Przyspiesza to wykonanie polecenia `netstat`.
Inną użyteczną opcją jest `--protocol=inet (lub –inet)`, która ogranicza wyniki tylko do gniazd protokołu IPv4.

Połącz się programem `ssh` z ze swoim komputerem. Adres IP znajdziesz wykonując komendę `ifconfig .`
`ssh student@XXX.XXX.XXX.XXX`

`ssh` łączy się używając protokołu TCP. Sprawdź poleceniem netstat stan połączenia ssh na komputerze pluton i komputerze PC. Rozłącz sesję ssh i sprawdź, czy pozostał ślad tej sesji.

Przetestuj opcję `–p` polecenia `netstat` z konta zwykłego użytkownika i z konta administratora (na PC). Jakie są różnice?

Ćwiczenie 2:

Skopiuj i skompiluj programy `udps.c` (serwer UDP) oraz `udpk.c` (klient UDP).

Uruchom serwer UDP. Upewnij się, na jakim porcie oczekuje na pakiety serwer.
Domyślnie jest to port nr 12900. Do każdego portu może się dowiązać tylko jedna aplikacja, co oznacza, że tylko jednej osobie uda się poprawnie uruchomić serwer na tym porcie.

Uruchom serwer UDP podając wybrany przez siebie numer portu XXX:
`./udps XXX`
Zobacz poleceniem `netstat` czy pojawiło się odpowiednie gniazdo. Na jakim interfejsie oczekuje na dane ten serwer?
Zalecane polecenie: `netstat --protocol=inet –an | grep XXX`

W drugim oknie uruchom klienta UDP podając jako adres serwera 127.0.0.1 port XXX. Prześlij jakieś dane.
Co raportuje serwer? Kto był nadawcą pakietów? Znajdź to gniazdo za pomocą `netstat`.

Zakończ proces klienta wpisując jako tekst do wysłania „quit”.

Uruchom ponownie klienta podając jako adres serwera 149.156.203.6 port XXX. Prześlij dane. Czy dane dotarły?

Zakończ klienta j.w. Zakończ serwer wysyłając mu sygnał.

Uruchom ponownie serwer UDP podając jako numer portu XXX oraz interfejs sieciowy 149.156.203.6:
`./udps XXX 149.156.203.6`

Zobacz poleceniem `netstat` czy pojawiło się odpowiednie gniazdo. Na jakim interfejsie oczekuje na dane ten serwer?

Uruchom klienta UDP podając jako adres serwera 149.156.203.6 port XXX. Prześlij jakieś dane. Czy dane dotarły? Zakończ proces klienta.

Uruchom klienta UDP podając jako adres serwera 127.0.0.1 port XXX. Prześlij jakieś dane. Czy dane dotarły? Zakończ proces klienta.
Zakończ proces serwera.

Ćwiczenie 3:

Skopiuj i skompiluj programy `tcps.c` (serwer TCP) oraz `tcpk.c` (klient TCP).

Uruchom serwer TCP podając jako argument wywołania wybrany przez siebie numer portu (XXX), na którym serwer będzie nasłuchiwał. W drugim oknie wykonaj polecenie:
`netstat –na | grep XXX`.

W drugim oknie uruchom klienta TCP, żądając połączenia na adres `149.156.203.6` i wybrany port `XXX`. Zaobserwuj ile jest teraz procesów `tcps`? Dlaczego więcej niż jeden? Co pokazuje `netstat`? Powinny być widoczne trzy gniazda. Dlaczego?

Zakończ proces klienta wpisując „quit”.

Połącz się z serwerem poprzez `telnet 149.156.203.6 XXX`. Prześlij jakieś dane. Obejrzyj gniazda. Ile ich powinno być i w jakim stanie?

Połącz się z serwerem TCP przez `telnet` z komputera lokalnego. Prześlij jakieś dane. Co teraz pokazuje `netstat`?

Zakończ wszystkie procesy klientów i proces serwera.

Ćwiczenie 4:

Napisz prosty serwer www, który na dowolne zapytanie odpowie zawartością strony www. Dla uproszczenia, nie będzie nas interesowało, jakiej strony żąda przeglądarka – zawsze wysyłamy tą samą. Dzięki temu nie musimy interpretować zapytania od przeglądarki.

Aby przeglądarka poprawnie zinterpretowała stronę należy w pierwszej kolejności wysłać ciąg znaków zgodny z protokołem http, który wygląda następująco:
`HTTP/1.0 200 OK\nContent-Type: text/html\n\n`

Następnie należy przesłać właściwą stronę www w postaci kolejnego ciągu znaków. Przygotowana strona testowa znajduje się w pliku site.html

Można posłużyć się programem tcps.c odpowiednio go modyfikując.

Laboratorium 10

- **Semafory**

1. Narzędzia IPC

Narzędzia IPC (ang. *Inter Process Communication*) służą do komunikacji pomiędzy procesami działającymi na jednym komputerze. Należą do nich:

- semafory
- pamięć dzielona (współdzielona)
- kolejki komunikatów

Semaforów używa się zwykle do synchronizacji procesów ze sobą, pamięć współdzielona jest używana do wymiany większych ilości danych, kolejki komunikatów są używane do przesyłania krótkich wiadomości oraz do synchronizacji procesów.

2. Definicja i warianty semaforów

Semafory są najczęściej stosowane do synchronizacji dostępu procesów do wspólnych zasobów (np. pamięci dzielonej, plików itp.).

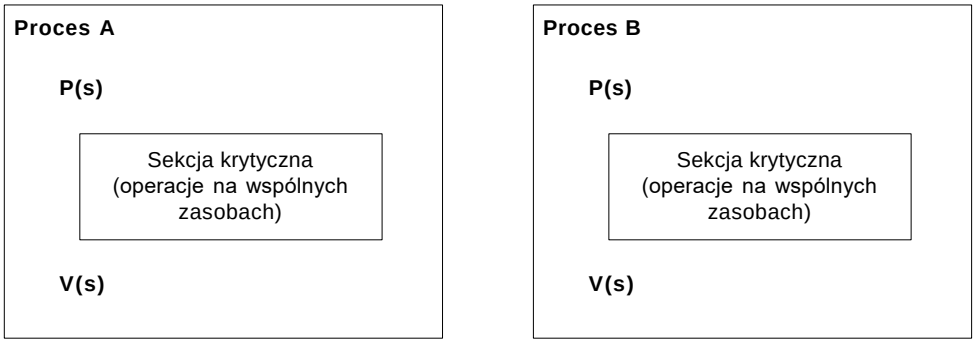
Podstawowym rodzajem semafora jest semafor binarny, przyjmujący dwa stany:

- opuszczony (zamknięty) – wówczas proces, który napotyka semafor musi zawiesić swoje działanie do momentu podniesienia semafora
- podniesiony (otwarty) – proces może kontynuować działanie

Oprócz semaforów binarnych istnieją również semafory wielostanowe.

W literaturze spotyka się konwencję, że funkcja *P(semafor)* służy do opuszczenia semafora a funkcja *V(semafor)* do jego podniesienia. Konwencję tę warto stosować niezależnie od języka programowania i sposobu realizacji semaforów. Należy zwrócić uwagę na to, że jeśli proces próbuje wykonać funkcję *P(semafor)* a semafor jest już opuszczony, to proces zostanie zawieszony do czasu podniesienia semafora przez inny proces.

Synchronizacja procesów korzystających ze wspólnego zasobu polega na zabezpieczeniu za pomocą semaforów fragmentu kodu, który wykonuje operacje na tym zasobie. Taki fragment kodu jest nazywany sekcją krytyczną. Jeżeli jeden proces wykonuje sekcję krytyczną, pozostałe zsynchronizowane procesy nie mogą jej wykonywać (Rys. 1)



Rys.1 Schemat synchronizacji dwóch procesów

W systemie UNIX zdefiniowany jest specjalny typ danych pozwalający zrealizować zarówno semafor wielostanowy jak też binarny. Pojedynczy semafor jest zmienną przyjmującą wartości całkowite nieujemne. Każdy semafor należy do ściśle określonego zbioru semaforów. Zbiór semaforów jest opisany przez rozbudowaną strukturę danych zawierającą między innymi informacje o prawach dostępu, czasach wykonania operacji itp. (patrz p. 4). W najprostszym przypadku, zbiór semaforów składa się z jednego semafora.
W systemie UNIX zdefiniowano jedynie strukturę semafora i zbiór funkcji do operacji na nich, bez określania jakie wartości semafora odpowiadają jakim jego stanom. Możliwe są dwie konwencje:

1. opuszczony ma wartość =0
   podniesiony ma wartość >0

2. opuszczony ma wartość >0
   podniesiony ma wartość =0

W przypadku konwencji pierwszej semafor może być wielokrotnie podniesiony (semaforowi opuszczonemu odpowiada tylko jeden stan), zaś w przypadku konwencji drugiej – odwrotnie.

3. Oglądanie semaforów

W celu wyświetlenia listy zbiorów semaforów aktualnie istniejących w systemie należy wykonać polecenie:
`> ipcs -s`
W wyniku wykonania polecenia otrzymujemy następujące informacje:

`KEY` – klucz, na podstawie którego utworzono zbiór semaforów
`SEMID` – unikalny identyfikator zbioru semaforów
`PERMS` – zawiera prawa dostępu do zbioru semaforów (można nadawać prawo czytania i modyfikacji).
`OWNER` – nazwa właściciela zbioru semaforów
`NSEMS` – liczba semaforów w zbiorze semaforów

W celu uzyskania dodatkowych informacji o zbiorach semaforów należy dodać opcje `-c`. Wyświetlane są wówczas następujące dodatkowe pola:

`GROUP` – nazwa grupy, do której należy zbiór semaforów
`CREATOR` – nazwa użytkownika, który utworzył zbiór semaforów
`CGROUP` – nazwa grupy – twórcy zbioru semaforów

Opcja `-t` wyswietla informacje o czasie:

`LAST-OP` – czas ostatniej operacji na zbiorze semaforów
`LAST-CHANGED` – czas utworzenia zbioru semaforów

Przykład 17

Poniżej przedstawiono przykładową listę zbiorów semaforów uzyskaną poleceniem `ipcs -s` i `ipcs -sc`:

```
> ipcs -s
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x4204c432 5439517    pbabicz    644        1
0x4204c552 5472287    acwiertni  644        1
0x420402c6 5570593    dwygonik   644        1
0x4204c50a 5636131    acurylo    644        1
0x4204c5e2 5668900    bgodlewsk  644        1
0x4204001a 5799976    mpolchlop  644        1
0x4204c5c7 5865514    plamch     644        1
0x420483f4 5963820    gpietrusz  644        1

> ipcs -sc
------ Semaphore Arrays Creators/Owners --------
semid      perms      cuid       cgid       uid        gid
5439517    644        pbabicz    eit2013    pbabicz    eit2013
5472287    644        acwiertnia eit2013    acwiertnia eit2013
5570593    644        dwygonik   eit2013    dwygonik   eit2013
5636131    644        acurylo    eit2013    acurylo    eit2013
5668900    644        bgodlewski eit2013    bgodlewski eit2013
5799976    644        mpolchlopek eit2013    mpolchlopek eit2013
5865514    644        plamch     stud2012   plamch     stud2012
5963820    644        gpietrusza stud2012   gpietrusza stud2012
```

W celu usunięcia zbioru semaforów z systemu można wykonać polecenie:
`> ipcrm -s ID`
gdzie `ID` jest identyfikatorem usuwanego zbioru semaforów.

4. Tworzenie unikalnego klucza

Do tworzenia wszystkich narzędzi IPC, w tym semaforów, potrzebny jest klucz. W większości systemów jest to liczba typu `int` (zwykle zdefiniowany jest typ `key_t`). Dzięki kluczom, do korzystania z tego samego narzędzia IPC przez wiele procesów, nie jest niezbędna znajomość identyfikatora narzędzia IPC. Identyfikator jest przydzielany przez system w momencie tworzenia narzędzia, a jego wartości nie da się przewidzieć. Natomiast procesy posługujące się tym samym kluczem do tworzenia narzędzia IPC (np. zbioru semaforów) mają gwarancję, że uzyskają dostęp do tego samego narzędzia.

Teoretycznie programista mógłby sobie samodzielnie wybrać klucz, jakim będą posługiwać się procesy współdziałające w ramach pisanej przez niego aplikacji. Przy takim założeniu groźba użycia tego samego klucza przez autorów różnych aplikacji jest znaczna.

Wyznaczanie unikalnego klucza wspomaga funkcja systemowa `ftok`. Klucz jest generowany na podstawie numeru węzła wskazanego pliku (pierwszy argument) oraz pojedynczego znaku (drugi argument). Prawdopodobieństwo powtórzenia się klucza jest tutaj niskie. Przyjęcie konwencji, że proces tworzący klucz podaje ścieżkę do jakiegoś pliku należącego do tej aplikacji minimalizuje prawdopodobieństwo powtórzenia klucza przez dwie różne aplikacje.

```
key_t ftok(const char *path, char proj)
```

Ścieżka `path` musi wskazywać na istniejący plik.

Przykład 18

Poniżej przedstawiono kod prostego programu `ft.c` generującego klucz na podstawie ścieżki do pliku „`.`” oraz znaku `A`.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>

main ()
{
   char          ID = 'A';          /* identyfikator do tw. semaf.  */
   char          path[20] = {"."};  /* plik do generowania klucza */
   key_t         klucz;             /* klucz do tworzenia IPC */
   klucz = ftok(path, ID);          /* stworz klucz dla pliku i ID */
   printf("--- klucz wg sciezki %s i ID %c =0x%x ---\n",path,ID,klucz);
 }
```

5. Funkcje do operacji na semaforach

Operacje podnoszenia i opuszczania semaforów muszą być wykonane w sposób niepodzielny, tzn. tylko jeden proces może w danej chwili sprawdzać i modyfikować stan semafora. W szczególności operacja opuszczenia semafora, wymagająca sprawdzenia, czy semafor jest podniesiony i następnie zmiany jego stanu na opuszczony musi być wykonana niepodzielnie. Funkcje systemowe UNIX'a zapewniają takie działanie.

a) Utworzenie zbioru semaforów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Funkcja `semget` tworzy w systemie **zbiór semaforów** lub dołącza się do istniejącego zbioru semaforów, następnie **zwraca identyfikator zbioru semaforów** (w przypadku błędu zwraca wartość -1).

Jeżeli zostanie podany klucz, dla którego istnieje już zbiór semaforów to wówczas funkcja będzie próbowała „dołączyć się do tego istniejącego zbioru". Zależy to jednak od ustawienia flag `IPC_CREAT`, `IPC_EXCL` (trzeci argument funkcji `semget`).

- Jeżeli nie ustawiono flagi `IPC_CREAT` wówczas funkcja `semget` będzie próbować dołączyć się do istniejącego zbioru semaforów. Jeżeli jednak nie istnieje zbiór semaforów utworzony na podstawie podanego klucza wówczas funkcja zwróci błąd. Wyjątkiem jest sytuacja, gdy podana wartość klucza jest równa `IPC_PRIVATE`, o czym będzie mowa dalej.
- Jeżeli ustawiono flagę `IPC_CREAT` i nie ustawiono `IPC_EXCL` wówczas funkcja `semget` utworzy nowy zbiór semaforów jeżeli wcześniej nie istniał zbiór utworzony na podstawie podanego klucza lub podłączy się do istniejącego zbioru semaforów.
- Jeżeli ustawiono `IPC_CREAT` i `IPC_EXCL` wówczas funkcja `semget` utworzy nowy zbiór semaforów na podstawie podanego klucza, jeżeli wcześniej taki zbiór nie istniał. Jeżeli jednak istnieje już zbiór semaforów utworzony na podstawie podanego klucza wówczas funkcja zwróci błąd

Zamiast klucza można podać flagę `IPC_PRIVATE`. Wówczas mamy gwarancję, że zostanie utworzony nowy, unikalny zbiór semaforów (ustawienie flag `IPC_CREAT`, `IPC_EXCL` nie ma znaczenia). Mamy również gwarancję, że żaden proces w przyszłości nie dołączy się do tego zbioru semaforów. Z tego względu flaga `IPC_PRIVATE` ma zastosowanie głównie dla aplikacji, w których jeden nadrzędny proces tworzy narzędzia IPC, zaś pozostałe dziedziczą identyfikatory tych narzędzi.
Zbiór semaforów będzie zawierał tyle pojedynczych semaforów ile wynosi wartość parametru `nsems`. Będą one indeksowane od wartości `0` do `nsems - 1`.

Trzeci argument funkcji `semget`, oprócz wspomnianych wcześniej flag `IPC_CREAT`, `IPC_EXCL`, służy do określenia praw dostępu do zbioru semaforów. Prawa dostępu definiuje się albo za pomocą liczby (w kodzie ósemkowym) albo za pomocą odpowiednich stałych symbolicznych.

Stała symboliczna	**wartość**	**znaczenie**
`SEM_R`	`0400`	prawo czytania dla użytkownika
`SEM_A`	`0200`	prawo modyfikacji dla użytkownika
`SEM_R >> 3`	`0040`	prawo czytania dla grupy
`SEM_A >> 3`	`0020`	prawo modyfikacji dla grupy
`SEM_R >> 6`	`0004`	prawo czytania dla pozostałych
`SEM_A >> 6`	`0002`	prawo modyfikacji dla pozostałych

Struktura danych semafora:

```
struct semid_ds {
     struct ipc_perm   sem_perm;    /* operation permission struct */
     struct sem        *sem_base;   /* ptr to first semaphore in set */
     ushort_t          sem_nsems;   /* number of sems in set */
     time_t            sem_otime;   /* last operation time */
     time_t            sem_ctime;   /* last change time */
}

struct ipc_perm {
     uid_t     uid;    /* owner user id */
     gid_t     gid;    /* owner group id */
     uid_t     cuid;   /* creator user id */
     gid_t     cgid;   /* creator group id */
     mode_t    mode;   /* r/a permission */
     ulong_t   seq;    /* slot usage sequence number */
     key_t     key;    /* key */
}

struct sem {
     ushort_t   semval;    /* semaphore value */
     pid_t      sempid;    /* pid of last operation  */
     ushort_t   semncnt;   /* # awaiting semval > cval */
     ushort_t   semzcnt;   /* # awaiting semval = 0 */
}
```

b) Wykonanie ciągu operacji na semaforach w sposób niepodzielny

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

Funkcja `semop` wykonuje ciąg operacji na zbiorze semaforów. Opisy operacji są przekazane przy pomocy wskaźnika `sops`. Jest to wskaźnik do pierwszego elementu tablicy zawierającej elementy postaci `struct sembuf`. W szczególności tablica może zawierać tylko jeden element, czyli opis jednej operacji. Liczba operacji przekazana jest przy pomocy paramteru `nsops`. Identyfikator zbioru semaforów, którego dotyczą operacje jest przekazany przy pomocy parametru `semid`. Opis pojedynczej operacji zawiera się w strukturze typu `struct sembuf`.

```
struct sembuf {
      int sem_num;
      int sem_op;
      int sem_flg;
}
```

Operacja określona przez pole `sem_op` jest wykonywana na semaforze określonym przez indeks `sem_num`. Pole `sem_op` może określić jedną z trzech rodzajów operacji na semaforze (w zależności od znaku):

1. `sem_op` ma wartość ujemną, wówczas:
   - Jeżeli aktualna wartość danego semafora jest większa bądź równa wartości bezwzględnej `sem_op`, to wartość aktualna semafora jest pomniejszana o wartość bezwzględną `sem_op`.
   - Jeżeli aktualna wartość danego semafora jest mniejsza od wartości bezwzględnej `sem_op`, to proces zawiesza wykonywanie działania do momentu gdy wartość semafora będzie większa bądź równa wartości bezwzględnej `sem_op`. Wznowienie procesu nastąpi też w razie usunięcia z systemu zbioru semaforów (w tym przypadku funkcja zwróci błąd), lub otrzymania sygnału.
   - Jeżeli aktualna wartość danego semafora jest mniejsza od wartości bezwzględnej `sem_op` i pole `sem_flg` jest ustawione na wartość `IPC_NOWAIT`, wówczas proces nie zawiesza wykonywania działania, natomiast funkcja zwraca błąd wykonania operacji (nie udało się ustawić żądanej wartości).
2. `sem_op` ma wartość dodatnią, wówczas:
   - Aktualna wartość danego semafora jest powiększana o wartość `sem_op`.
3. `sem_op` ma wartość zero, wówczas:
   - Jeżeli aktualna wartość danego semafora jest równa zero, funkcja powraca natychmiast,
   - Jeżeli aktualna wartość danego semafora jest różna od zera, funkcja zawiesza działanie procesu do czasu, gdy wartość semafora zostanie ustawiona na zero, zbiór semaforów zostanie usunięty z systemu, lub proces otrzyma sygnał.
   - Jeżeli aktualna wartość danego semafora jest różna od zera i pole `sem_flg` jest ustawione na wartość `IPC_NOWAIT`, wówczas proces nie zawiesza wykonywania działania, natomiast funkcja zwraca błąd wykonania operacji.

W polu `sem_flg` może być również ustawiona flaga `SEM_UNDO`. Operacje wykonane z tą flagą zostają automatycznie odwołane w momencie zakończenia procesu. Jest to istotne w sytuacji, w której proces zakończy działanie będąc w sekcji krytycznej bez podniesienia semaforów, co normalnie (bez flagi `SEM_UNDO`) spowodowałoby zablokowanie działania pozostałych zsynchronizowanych procesów.

W przypadku poprawnie wykonanej operacji funkcja `semop` zwraca wartość zero. W przypadku błędu funkcja zwraca wartość –1.

Przykładowe zastosowanie:

	podejście 1 (0-semafor opuszczony)		**podejście 2 (0-semafor podniesiony)**			
operacja:	- opuść semafor (jeżeli opuszczony to czekaj aż ktoś go podniesie i	- podnieś	- opuść semafor (jeżeli opuszczony to czekaj aż ktoś go podniesie i	- opuść	- czkaj aż ktoś podniesie	- podnieś

	natychmiast opuść)		natychmiast opuść)			
`nsops`	`1`	`1`	`2`	`1`	`1`	`1`
`sem_op`	`-1`	`+1`	`0` `+1`	`+1`	`0`	`-1`

c) Operacje kontrolne na zbiorze semaforów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun);

union semun {
             int val;
             struct semid_ds *buf;
             ushort_t *array;
 } arg ;
```

Funkcja `semctl` umożliwia realizację operacji kontrolnych na zbiorach semaforów. Dzięki niej można między innymi **usunąć z systemu zbiór semaforów**, ustawiać wartości poszczególnych semaforów w zbiorze, lub je odczytywać. Można również zmieniać uprawnienia procesów w zakresie dostępu do zbioru semaforów. Parametr `semid` określa identyfikator zbioru semaforów (zwrócony przez funkcję `semget`); `semnum` określa indeks semafora, którego dotyczy dana operacja. Parametr `cmd` określa rodzaj operacji kontrolnej, którą chcemy wykonać na danym semaforze. Ostatni parametr zależy od rodzaju komendy i zawiera niezbędne dane do jej wykonania. Najważniejsze operacje kontrolne:

- `GETVAL` - zwraca wartość semafora (o ile proces ma uprawnienia do odczytu wartości), czwarty parametr nie ma znaczenia dla tej operacji, wartość semafora jest zwracana jako wynik działania funkcji,
- `SETVAL` - ustawia semafor na wartość określoną czwartym parametrem `val` (o ile proces ma uprawnienia do modyfikacji wartości),
- `GETALL` - zwraca wartości wszystkich semaforów w zbiorze, wpisując je do tablicy `array`,
- `SETALL` - ustawia wszystkie semafory przypisując im wartości z tablicy `array`,
- `IPC_STAT` - pobiera informację o statusie zbioru semaforów i wpisuje ją do struktury wskazywanej przez parametr `buf`,
- `IPC_RMID` - **usuwa zbiór semaforów** z systemu i likwiduje strukturę danych opisujących tablicę semaforów. Zbiór semaforów może zostać usunięty przez proces mający efektywny identyfikator użytkownika taki sam jak identyfikator właściciela bądź twórcy danego zbioru semaforów.

Należy wyraźnie podkreślić, że funkcja `semctl` NIE MOŻE być używana do zabezpieczania sekcji krytycznej w procesie (zamiast funkcji `semop`), gdyż nie gwarantuje niepodzielności operacji.

6. Pytania kontrolne

1. Wymień narzędzia IPC
2. Co to jest semafor?
3. Co to jest semafor binarny?
4. Do czego służą funkcje *P(s)* i *V(s)*?
5. Semafor binarny jest opuszczony. Została wywołana funkcja *P(s)* służąca do opuszczenia semafora. Co się stanie?
6. Dlaczego warto tworzyć klucz funkcją `ftok`?
7. Jakim poleceniem wyświetlić listę utworzonych w systemie zbiorów semaforów?
8. Co się stanie z innymi zsynchronizowanymi procesami, gdy proces nagle zakończy się w sekcji krytycznej?
9. Dlaczego nie należy używać funkcji `semctl` do realizacji funkcji *P(s)* i *V(s)*
10. Kto może usunąć zbiór semaforów?
11. Do czego służy funkcja `semctl`?

7. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl`
Programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB10`

Uwaga !! Po skończonych zajęciach proszę usunąć pozostałe zbiory semaforów

Ćwiczenie 1:

Skopiuj i skompiluj programy `sem_w.c` oraz `sem_up.c`. Programy opatrzone zostały licznymi komentarzami. Należy na nie zwrócić uwage analizując kod. Przyjęto konwencję, że semafor podniesiony ma wartość równą 0. Program sem_w.c tworzy zbiór semaforów złożony z jednego semafora. Semafor jest inicjalizowany jako opuszczony. W związku z tym proces zawiesza swoje działanie w oczekiwaniu na podniesienie semafora. Do podniesienia semafora służy program sem_up.c.

a) Uruchom w na jednym terminalu program sem_w. Obejrzyj listę wszystkich narzędzi IPC oraz listę samych semaforów. Poszukaj utworzonego prze siebie zbioru semaforów.

- Jakie ma parametry?
- Jaki jest teraz czas ostatniej operacji?
- Ile jest semaforów w zbiorze?
- Jakie są prawa dostępu, identyfikator, klucz?

Dla swojego semafor przetestuj opcję `-i` polecenia `ipcs`. Na drugim terminalu uruchom program sem_up. Zaobserwuj działanie obu procesów. Jaki jest teraz czas ostatniej operacji?

b) Uruchom na jednym terminalu sem_w pięć razy w tle. Utworzonych zostanie pięć procesów. Zwróć uwagę na numery PID uruchamianych procesów. Z drugiego terminala uruchom sem_up odpowiednią liczbę razy tak, aby wszystkie procesy sem_w weszły do swoich sekcji krytycznych. Zaobserwuj w jakiej kolejności procesy sem_w otrzymywały dostęp do sekcji krytycznej. Czy w tej samej, w której zostały uruchomione?

Po skończonym ćwiczeniu usuń utworzony przez siebie zbiór semaforów za pomocą polecenia ipcrm.

Nie jest utrzymywana „kolejka do semafora". Procesy nie są dopuszczane do sekcji krytycznej w kolejności „zgłoszenia się do semafora" lecz w kolejności losowej. Do sekcji krytycznej wejdzie ten z oczekujących procesów, który jako pierwszy otrzyma dostęp do procesora. Należy o tym pamiętać pisząc aplikacje, w których procesy cyklicznie wchodzą do swoich sekcji krytycznych. Jeden semafor binarny może wówczas nie wystarczyć.

Ćwiczenie 2:

Skopiuj pliki sem.h oraz sem_pv.c. Program sem_pv.c po uruchomieniu tworzy zbiór semaforów, a następnie cyklicznie wchodzi do swojej sekcji krytycznej. Zastosowano konwencję, że semafor podniesiony ma wartość większą od 0. Plik nagłówkowy sem.h zawiera definicję kilku funkcji, w tym funkcji *P()* i *V()*.

- Uruchom sem_pv w tle. Zaobserwuj jego działanie.
- Uruchom po raz drugi sem_pv w tle (można to zrobić na drugim terminalu). Teraz 2 procesy powinny wykonywać swoje sekcje krytyczne na przemian.
- Zobacz co się stanie, gdy zakończysz jeden z procesów (wysyłając sygnał) w momencie gdy jest w swojej sekcji krytycznej.
- Czy drugi proces nadal działa (wykonuje cyklicznie swoją sekcje krytyczną)? Co należałoby zmienić, aby taka sytuacja nie wystąpiła?
- Zakończ drugi proces.
- Uruchom ponownie dwa procesy jednocześnie, używając polcenia `./sem_pv & ./sem_pv &`. Czy procesy nadal poprawnie się synchronizują? Usuń zbiór semaforów poleceniem ipcrm. Jak zareagowały procesy?

Ćwiczenie 3:

Skopiuj pliki sem_wrong.h oraz sem_pv_wrong.c. Z założenia program ten powinien działać dokładnie tak samo jak program w ćwiczenia 2. Różnica polega na tym, że funkcje P() i V() zostały napisane błędnie (operacje na semaforze są wykonywane za pomocą funkcji semctl zamiast semop.

- Uruchom ponownie dwa procesy jednocześnie, używając polcenia `./sem_pv_wrong & ./sem_pv_wrong &`. Czy procesy poprawnie się synchronizują?

Ćwiczenie 4:

Skopiuj programy `count1.c` oraz `count2.c` i skompiluj. Pierwszy z nich wyświetla na ekranie liczby nieparzyste w następujący sposób:

```
1
3 3 3
5 5 5 5 5
```

itd. Po każdej wyświetlonej linii proces zasypia na czas 1 sekundy. Program `count2.c` wyświetla liczby parzyste w następujący sposób:

```
2 2
4 4 4 4
6 6 6 6 6 6
```

itd. Po każdej wyświetlonej liczbie proces zasypia na czas 1 sekundy. Jeśli uruchomimy te programy jednocześnie (`./count1 & ./count2 &`), wówczas wyniki ich wykonania będą na ekranie przemieszane.

Należy zsynchronizować te procesy za pomocą semaforów tak, aby otrzymać na ekranie następujący obraz:

```
1
2 2
3 3 3
4 4 4 4
```

itd. W obu programach oznaczono początek i koniec sekcji krytycznej. Kodu sekcji krytycznej zmieniać nie wolno. W pozostałej części programów można dopisać niezbędne operacje na semaforach. Zabronione jest stosowanie gdziekolwiek dodatkowych funkcji `sleep`. Wybór konwencji jest dowolny. Programy mają działać poprawnie niezależnie od tego w jakiej kolejności i jakim odstępie czasowym zostaną uruchomione. Liczbę semaforów i sposób implementacji należy dobrać tak, aby nie zachodziło niebezpieczeństwo, że jeden z procesów będzie cyklicznie wchodził do swojej sekcji krytycznej, a drugi będzie w nieskończoność czekał.

Laboratorium 11

- **Pamięć współdzielona**

1. Wprowadzenie

System UNIX chroni obszar pamięci danych każdego procesu. Wejście w obszar danych innego procesu powoduje błąd adresacji i przerwanie pracy procesu.

Dla komunikacji pomiędzy procesami można stworzyć pewien obszar w pamięci operacyjnej wspólny dla współpracujących procesów. Umożliwia to efektywne przesyłanie danych pomiędzy procesami. Jeden proces może utworzyć segment pamięci współdzielonej, a następnie inne procesy mogą się do niego dołączać i z niego korzystać. Operacje na tej pamięci odbywają się jak na zwykłej pamięci procesu. Struktura danych przechowywanych w segmencie pamięci współdzielonej może być dowolna.

Istotnym problemem jest zapewnienie synchronizacji pomiędzy procesami korzystającymi z pamięci współdzielonej. Jeżeli więcej niż jeden proces zapisuje dane do pamięci współdzielonej może to powodować konflikty i wzajemne nadpisywanie danych przekazywanych przez różne procesy. Zespół operacji na pamięci współdzielonej powinien być traktowany jako sekcja krytyczna. Synchronizację procesów można zapewnić za pomocą semaforów lub kolejek komunikatów.

2. Oglądanie pamięci współdzielonych

W celu wyświetlenia listy segmentów pamięci współdzielonej aktualnie istniejących w systemie należy wykonać polecenie:

```
> ipcs -m
```

W celu uzyskania pełnego opisu segmentów pamięci współdzielonej należy dodać dodatkowe opcje (patrz polecenie: `man ipcs`). Następujące informacje o pamięci dzielonej są dostępne przy użyciu polecenia `ipcs`:

`ID`	– unikalny identyfikator segmentu pamięci współdzielonej
`KEY`	– klucz, na podstawie którego utworzono segment pamięci współdzielonej
`MODE`	– zawiera prawa dostępu do pamięci współdzielonej (można nadawać prawo czytania i pisania). Ponadto, jako pierwszy znak może wystąpić litera `D` (patrz opis poniżej)
`OWNER`	– nazwa właściciela pamięci współdzielonej
`GROUP`	– nazwa grupy, do której należy pamięć współdzielona
`CREATOR`	– nazwa użytkownika, który utworzył pamięć współdzieloną
`CGROUP`	– nazwa grupy, do której należy twórca pamięci współdzielonej
`NATTCH`	– liczba procesów dołączonych do segmentu pamięci współdzielonej
`SEGSZ`	– rozmiar segmentu pamięci współdzielonej
`CPID`	– PID procesu, który utworzył pamięć współdzieloną
`LPID`	– PID procesu, który ostatnio dołączył się lub odłączył od pamięci współdzielonej
`ATIME`	– czas ostatniego dołączenia się jakiegoś procesu do pamięci współdzielonej
`DTIME`	– czas ostatniego odłączenia się jakiegoś procesu od pamięci współdzielonej
`CTIME`	– czas utworzenia segmentu pamięci współdzielonej

Pamięć współdzielona może być skasowana natychmiast, jeżeli żaden proces nie jest do niej dołączony. Jeżeli istnieją procesy dołączone do pamięci współdzielonej, wówczas nie jest ona kasowana natychmiast, ale dopiero w momencie odłączenia się ostatniego procesu. W takiej sytuacji klucz, na podstawie którego została utworzona pamięć współdzielona jest zerowany. Do takiego segmentu pamięci nie może już dołączyć się żaden nowy proces. W zależności od wersji systemu skasowana w ten sposób pamięć współdzielona albo staje się niewidoczna dla polecenia `ipcs` albo jest wyświetlana, ale w polu `MODE` pojawia się litera `D`.

Jeżeli żaden proces nie wykonywał operacji dołączania lub odłączania się od pamięci współdzielonej wówczas czasy `ATIME` i `DTIME` mogą mieć wartość `no-entry`.

Przykład 19

Poniżej przedstawiono przykładową listę segmentów pamięci współdzielonej uzyskaną poleceniem `ipcs -m`:

```
> ipcs -m

------ Shared Memory Segments --------
key        shmid      owner     perms     bytes     nattch     status
0x00000000 5603330    root      644        80         2
0x00000000 5636100    root      644        16384      2
0x00000000 5668869    root      644        280        2
```

I szczegółowe informacje o jednym segmencie:

```
> ipcs -m -i 5603330

Shared memory Segment shmid=5603330
uid=0  gid=0  cuid=0 cgid=0
mode=0644    access_perms=0644
bytes=80     lpid=3617    cpid=3610    nattch=2
att_time=Mon Dec 22 11:49:17 2014
det_time=Mon Dec 22 11:49:17 2014
change_time=Mon Dec 22 11:49:17 2014
```

W celu usunięcia pamięci współdzielonej z systemu można wykonać polecenie:
`> ipcrm -m ID`
gdzie `ID` jest identyfikatorem usuwanej pamięci współdzielonej.

3. Funkcje do operacji na pamięci współdzielonej

a) utworzenie segmentu pamięci współdzielonej

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

Funkcja `shmget` zwraca identyfikator istniejącego lub nowo utworzonego segmentu pamięci współdzielonej związanego z podanym kluczem. Należy podkreślić, że jest to jedynie uzyskanie identyfikatora narzędzia IPC. Aby proces mógł korzystać z segmentu pamięci współdzielonej musi uzyskać jego adres w pamięci, co zwane jest dołączaniem do segmentu pamięci współdzielonej, a realizowane przez funkcję `shmat` (punkt b).

Segment pamięci współdzielonej jest tworzony na podstawie podanego klucza. Zasady stosowania unikalnego klucza oraz flagi `IPC_PRIVATE` są takie same jak dla innych narzędzi IPC. Trzeci argument funkcji `shmget` określa sposób tworzenia pamięci współdzielonej i prawa dostępu. Podobnie jak dla innych narzędzi IPC można tu stosować flagi `IPC_CREAT` oraz `IPC_EXCL`. Analogicznie definiuje się też prawa dostępu.

Jeżeli nie istnieje segment pamięci współdzielonej i pozwalają na to ustawienia flag, funkcja `shmget` tworzy w systemie strukturę danych `shmid_ds` reprezentującą segment pamięci współdzielonej i rezerwuje w pamięci obszar o rozmiarze określonym przez argument `size`. Należy podkreślić, że w momencie tworzenia segmentu pamięci współdzielonej nie określa się struktury danych, które będą w nim przechowywane a jedynie rozmiar segmentu w bajtach. Faktyczną strukturę segmentu (sposób interpretacji poszczególnych bajtów) określa proces w momencie dołączania się do segmentu pamięci współdzielonej. W strukturze `shmid_ds` znajdują się atrybuty pamięci współdzielonej (te same, które można wyświetlić poleceniem `ipcs -ma`).

Funkcja `shmget` zwraca identyfikator pamięci współdzielonej. W przypadku błędu funkcja `shmget` zwraca wartość `-1` i wpisuje kod błędu do globalnej zmiennej `errno`.

b) dołączenie procesu do segmentu pamięci współdzielonej

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

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Po wywołani funkcji `shmget`, proces zna identyfikator segmentu pamięci współdzielonej. Nie jest mu jednak znany adres (wskaźnik) tego segmentu. Funkcja `shmat` umożliwia uzyskanie tego adresu (dołączenie procesu do segmentu pamięci współdzielonej). Dzięki tej funkcji proces otrzymuje wskaźnik na pierwszy bajt pamięci wspólnej. Pierwszy argument określa identyfikator segmentu pamięci współdzielonej, do którego wskaźnik chcemy uzyskać. Drugi określa adres z przestrzeni adresowej dostępnej dla procesu, pod którym ma być przyłączona pamięć współdzielona. Jeżeli wartość tego argumentu wynosi `NULL`, wówczas system operacyjny samodzielnie dobiera odpowiednią wartość i zwraca ją jako wynik działania funkcji. Jeżeli wartość adresu podanego przez użytkownika jest niezerowa, wówczas, w zależności od ustawienia flagi `SHM_RND` (trzeci argument) system w sposób elastyczny bądź ścisły przyłącza segment pamięci współdzielonej pod adres żądany przez proces. Takimi przypadkami jednak nie będziemy się zajmować, gdyż w praktyce są bardzo rzadko stosowane.

W przypadku pomyślnego dołączenia do segmentu pamięci współdzielonej, funkcja zwraca wskaźnik do początku współdzielonego obszaru. Zwracany wskaźnik ma typ nieokreślony (`void`). Aby poprawnie interpretować poszczególne bajty w segmencie pamięci wspólnej proces dołączający musi zrzutować ten wskaźnik na odpowiedni typ danych. Poprzez takie rzutowanie niejako narzucamy sposób interpretacji zawartości pamięci współdzielonej. Programista może dowolnie określić, co jest przechowywane w pamięci współdzielonej, ważne jednak jest, aby wszystkie procesy korzystające z tego samego segmentu pamięci współdzielonej interpretowały go w sposób jednolity.

W przypadku wystąpienia błędów zwracana jest wartość `-1`. Ponieważ zwracany typ to wskaźnik, do prawidłowej kontroli błędów, należy zwróconą wartość zrzutować na typ `int` i porównać ze stałą `-1`. Jeżeli wartość ta jest różna od `-1`, to znaczy, że segment pamięci współdzielonej został prawidłowo dołączony i można z niego korzystać (patrz przykład 20).

c) odłączenie procesu od segmentu pamięci współdzielonej

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

int shmdt(char *shmaddr);
```

Funkcja `shmdt` umożliwia odłączenie procesu od utworzonego wcześniej przez ten lub inny proces segmentu pamięci współdzielonej, do którego dany proces jest dołączony. Argument funkcji jest wskaźnikiem do segmentu pamięci współdzielonej. W przypadku pomyślnego wykonania operacji funkcja zwraca wartość zero. W przypadku błędu funkcja zwraca wartość `-1`, a kod błędu wpisywany jest do zmiennej globalnej `errno`.

d) operacje kontrolne na pamięci współdzielonej

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Funkcja `shmctl` umożliwia realizację operacji kontrolnych na pamięci współdzielonej. Dzięki niej można między innymi **usunąć z systemu segment pamięci współdzielonej**. Można również zmieniać uprawnienia procesów w zakresie dostępu do pamięci współdzielonej. Argument `shmid` określa identyfikator segmentu pamięci. Argument `cmd` określa rodzaj operacji kontrolnej, która ma być wykonana na pamięci współdzielonej. Ostatni argument zależy od rodzaju komendy i zawiera niezbędne dane do jej wykonania.

Najważniejsze operacje kontrolne:

- IPC_STAT - pobiera informację o statusie segmentu pamięci współdzielonej i wpisuje ją do struktury wskazywanej przez argument `buf`,
- IPC_RMID - **usuwa segment pamięci współdzielonej** z systemu i likwiduje strukturę danych opisujących ten segment. Segment pamięci współdzielonej może usunąć proces, który go utworzył, albo proces, który ma do tego uprawnienia.

W przypadku błędu funkcja zwraca wartość `-1` i wpisuje kod błędu do globalnej zmiennej `errno`.

Przykład 20

Poniżej zaprezentowano przykładowy program obsługujący pamięć współdzieloną. Zdefiniowano strukturę danych jakie będą przechowywane w pamięci współdzielonej. Następnie wykonywane są kolejno następujące operacje:

- utworzenie segmentu pamięci współdzielonej,
- dołączenie segmentu do procesu,
- zapis danych do pamięci współdzielonej,
- odczyt danych,
- odłączenie segmentu pamięci współdzielonej,
- skasowanie segmentu pamięci współdzielonej.

Po każdej operacji program zawiesza działanie na 10 s. W przypadku tworzenia, dołączania, odłączania i kasowania sprawdzana jest poprawność wykonania tych operacji. Jeżeli wystąpił błąd program kończy działanie.

shm.c

```
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>


main(){

   int shmID;                                  /* id segmentu pamięci współdzielonej*/
   struct Dane {                               /* taka będzie struktura danych segmentu */
       int numer;
       int wartosc;
       char napis[10];
       time_t czas;
   } *dane_ptr;

   key t klucz = ftok(".",'K');                /* utworzenie segmentu pamięci współdzielonej z
                                                  prawami dostępu rw-r----- i rozmiarze
                                                  rownym rozmiarowi struktury danych, ktora
                                                  ma byc przechowywana w segmencie.*/


   shmID=shmget(klucz,sizeof(struct Dane), IPC CREAT | SHM R  | SHM W  | SHM R >> 3);

                                               /* czy udało się utworzyć pamięć wspoldziel.?*/
   if(shmID == -1){
       perror("Blad utworzenia/dolaczenia identyfikatora pamieci wspoldzielonej");
       return 1;
   }

   printf("Utworzono pamiec wspoldzielona o ID=%d\n",shmID);
   sleep(10);

                                               /* dołączenie segmentu pamięci współdzielonej i
                                                  rzutowanie wskaźnika na „nasz" typ danych
                                                  przechowywanych w pamięci współdzielonej
                                                  (struktura typu Dane) */

   dane ptr = (struct Dane *) shmat(shmID, 0, 0);

                                               /* czy dołączenie się powiodło? */
   if(((int )dane_ptr)==-1){
       perror("Blad dolaczenia");
       return 2;
   }
```

```
   printf("Dolaczono strukture danych do obszaru pamieci wspoldzielonej\n");
   sleep(10);

                                                /* zapisanie danych do segmentu pamięci
                                                   współdzielonej */
   dane ptr->numer = 2;
   dane_ptr->wartosc = 5;
   strcpy(dane_ptr->napis,"TEXT");
   dane_ptr->czas = time(NULL);


                                                /* odczyt danych z pamięci współdzielonej */

   printf("\nZAWARTOSC PAMIECI WSPOLDZIELONEJ:\nNumer %d\nWartosc %d\nNapis %s\nCzas
             %s\n",dane_ptr->numer,dane_ptr->wartosc,&dane_ptr->napis,ctime(&dane_ptr->czas));
   sleep(10);

                                                /* odłączenie segmentu pamięci współdzielonej i
                                                   sprawdzenie czy to się udało */
   if(shmdt((char *) dane_ptr) == -1) {
       perror("Blad przy odlaczaniu pamieci wspoldzielonej");
       return 3;
   }

   printf("Pamiec wspoldzielona odlaczona\n");
   sleep(10);

                                                /* skasowanie segmentu pamięci wspoldzielonej i
                                                   sprawdzenie czy to się udało */
   if(shmctl(shmID, IPC_RMID, 0) == -1) {
       perror("Blad przy usuwaniu pamieci wspoldzielonej");
       return 4;
   }

   printf("Pamiec wspoldzielona skasowana\n");
   sleep(10);

   return 0;
}
```

4. Pytania kontrolne

1. Do czego służy pamięć współdzielona?
2. Co może zawierać pamięć współdzielona?
3. Do czego służą funkcje `shmat` i `shmdt`?
4. W jaki sposób sprawdzić, kiedy i jaki proces ostatnio dołączył się bądź odłączył od segmentu pamięci współdzielonej?
5. W jaki sposób sprawdzić, kto utworzył segment pamięci współdzielonej?
6. Jak sprawdzić ile jest aktualnie procesów dołączonych do pamięci współdzielonej?
7. Czy pamięć współdzielona jest usuwana automatycznie, gdy skończy działanie proces, który ją stworzył?
8. Czy wymuszone usunięcie pamięci (funkcją `shmctl` lub poleceniem `ipcrm`) powoduje zawsze natychmiastowe usunięcie segmentu pamięci współdzielonej?
9. Czy możliwa jest sytuacja, że proces nie może dołączyć się do segmentu pamięci współdzielonej, mimo że używa tego samego klucza, według którego została utworzona i ma do tego odpowiednie prawa?

5. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl`
Programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB11`

Uwaga !! Po skończonych zajęciach proszę usunąć pozostałe zbiory semaforów i segmenty pamięci współdzielonej!!

Ćwiczenie 1:

Skopiuj programy `shm.c` i skompiluj. Kod tego programu znajduje się również w materiałach. Program wykonuje następujące operacje w odstępach 10 sekund:

- Uzyskanie identyfikatora segmentu pamięci współdzielonej (`shmget`)
- Dołączenie procesu do pamięci współdzielonej (`shmat`)
- Zapis i odczyt pamięci współdzielonej
- Odłączenie pamięci współdzielonej (`shmdt`)
- Skasowanie pamięci współdzielonej (`shmctl`)

Uruchom program i zaobserwuj jego działanie. Po każdej w wyżej wymienionych operacji obejrzyj parametry pamięci współdzielonej. Zwróć uwagę na liczbę procesów dołączonych, PID procesu, który utworzył segment pamięci współdzielonej oraz PID procesu, który ostatnio dołączył się bądź odłączył od pamięci współdzielonej oraz czasy ATIME, DTIME i CTIME.

- Jakie są wartości parametrów na początku (tuż po utworzeniu pamięci współdzielonej)? Jak zmieniają się później?
- Skąd wynika rozmiar pamięci współdzielonej równy 24B? Spróbuj samodzielnie policzyć jaki powinien być ten rozmiar. Czy jest rozmiar raportowany przez `ipcs` zgadza się z Twoimi przewidywaniami?

Ćwiczenie 2:

- Uruchom ponownie `shm` w tle. Usuń pamięć współdzieloną poleceniem `ipcrm` zaraz po utworzeniu (po zobaczeniu komunikatu podającego jej identyfikator). Czy powiedzie się dołączenie procesu do pamięci współdzielonej?
- Uruchom ponownie `shm` w tle. Usuń pamięć współdzieloną poleceniem `ipcrm` zaraz po dołączeniu się procesu do pamięci współdzielonej. Czy powiodą się operacje zapisu i odczytu pamięci współdzielonej? Sprawdź czy `ipcs` pokazuje Twój segment pamięci współdzielonej. Czy proces wykona operację odłączenia pamięci współdzielonej?
- Uruchom ponownie `shm` w tle. Usuń pamięć współdzieloną poleceniem `ipcrm` zaraz odłączeniu się procesu od pamięci współdzielonej. Na jakiej operacji teraz wystąpi błąd?

Ćwiczenie 3:

Napisz aplikację złożoną z dwóch programów: producenta i konsumenta. Procesy będą wymieniać dane poprzez segment pamięci współdzielonej. Producent zapisuje dane, a konsument odczytuje. Procesy te należy zsynchronizować za pomocą semaforów.
Struktura danych przechowywana w pamięci współdzielonej może być dowolna, powinna jednak zawierać przynajmniej jeden `int` i tablicę znaków. Dane do wpisania do pamięci współdzielonej producent pobiera z `stdin`. Po skompletowaniu danych w pamięci współdzielonej zezwala na ich odczyt przez konsumenta. Konsument wyświetla na ekranie odczytane dane, po czym zezwala producentowi na zapis nowej porcji danych, itd.

Laboratorium 12

- **Kolejki komunikatów**

1. Wprowadzenie

Kolejki komunikatów (*Message Queues*) są jednym z narzędzi IPC. Wszystkie kolejki są pamiętane w jądrze systemu i mają przypisane identyfikatory kolejki. Procesy mogą czytać i zapisywać komunikaty do różnych kolejek. Każdy komunikat w kolejce ma następujące atrybuty:

- **typ** komunikatu (liczba całkowita)
- **długość** porcji danych przesyłanych w komunikacie (może być równa 0)
- **dane** (jeżeli długość jest większa od zera)

W strukturze komunikatu nie ma „adresata" komunikatu. Oznacza to, że nie można np. wysłać komunikatu wprost do procesu o określonym PID. Komunikujące się ze sobą procesy powinny korzystać z tych samych kolejek oraz mieć „uzgodnione" znaczenie poszczególnych typów komunikatów.

Proces może umieścić komunikat w kolejce niezależnie od tego czy istnieje inny proces oczekujący na ten komunikat. Każdy komunikat jest przechowywany w kolejce aż do momentu, kiedy jakiś proces go odczyta (odczyt komunikatu powoduje jego usunięcie z kolejki) lub do momentu usunięcia kolejki z systemu. Komunikaty są przechowywane w kolejce również, gdy proces, który je wysłał zakończył się.

Przesyłanie komunikatów można wykorzystywać w aplikacjach klient–serwer oraz do synchronizacji procesów korzystających ze wspólnych zasobów (ochrona sekcji krytycznej).

2. Oglądanie kolejek komunikatów

W celu wyświetlenia listy kolejek komunikatów aktualnie istniejących w systemie należy wykonać polecenie:
```
> ipcs -q
```

W celu uzyskania pełnego opisu kolejek należy dodać dodatkowe opcje (`-c, -i, -p, -t, -u, -l`). Dostępne są wówczas następujące pola:

`ID`	– unikalny identyfikator kolejki
`KEY`	– klucz na podstawie którego utworzono kolejkę
`MODE`	– zawiera prawa dostępu do kolejki do komunikatów (można nadawać prawo czytania i pisania). Ponadto, jako drugi znak może wystąpić litera `R` (co oznacza, że jakiś proces oczekuje na możliwość odczytania komunikatu z tej kolejki) lub litera `S` (co oznacza, że jakiś proces oczekuje na możliwość wysłania komunikatu do kolejki). Patrz też przykład 17 oraz opisy funkcji `msgsnd` oraz `msgrcv`.
`OWNER`	– nazwa właściciela kolejki
`GROUP`	– nazwa grupy do której należy właściciel kolejki
`CREATOR`	– nazwa użytkownika, który utworzył kolejkę
`CGROUP`	– nazwa grupy użytkownika, który utworzył kolejkę
`CBYTES`	– aktualna liczba bajtów w kolejce (suma długości wszystkich komuniaktów)
`QNUM`	– aktualna liczba komunikatów w kolejce
`QBYTES`	– maksymalna liczba bajtów w kolejce
`STIME`	– czas ostatniego wysłania danych
`RTIME`	– czas ostatniego odbioru danych
`CTIME`	– czas utworzenia kolejki
`LSPID`	– PID ostatniego procesu, który zapisał komunikat do tej kolejki
`LRPID`	– PID ostatniego procesu, który odczytał komunikat z tej kolejki

Przykład 21

Wykonano polecenie:

```
> ipcs -q
IPC status from /dev/mem as of Sun Jan 25 14:02:56 NFT 1970
T    ID        KEY        MODE   OWNER    GROUP  CREATOR   CGROUP CBYTES  QNUM QBYTES LSPID LRPID    STIME    RTIME    CTIME
Message Queues:
q    50 0x51004ad2 --rw------- rstankie   staff rstankie    staff      0     0   4096     0     0 no-entry no-entry 10:03:27
q    54 0x5133d8e1 --rw-rw-rw- rstankie   staff rstankie    staff     19     2   4096   596     0 11:20:23 no-entry 11:20:20
q    58 0x513ae451 -Rrw-r--r-- rstankie   staff rstankie    staff      0     0   4096   612   705 11:22:00 11:28:43 11:21:12
```

- Kolejka o identyfikatorze `ID=50` została utworzona o godzinie `10:03:27`. Żaden proces nie zapisywał ani nie odczytywał komunikatów z tej kolejki (`LSPID` oraz `LRPID` są równe zero a czasy `STIME` i `RTIME` mają są ustawione na `no-entry`).
- Do kolejki o identyfikatorze `ID=54` utworzonej o godzinie `11:20:20` został ostatnio zapisany komunikat o godzinie `11:20:23` przez proces o `PID=596`. Liczba komuniaktów kolejce wynosi 2, zaś sumaryczna długość tych dwóch komunikatów to 19 bajów.
- Do kolejki o `ID=58` ostatnio zapisał komunikat proces o `PID=612` o godzinie `11:22:00`, zaś proces o `PID=705` odczytał komunikat o godzinie `11:28:43`. Ponadto jakiś proces oczekuje na odczyt komunikatu z kolejki (`MODE=-Rrw-r--r--`).

W celu usunięcia kolejki komunikatów z systemu można wykonać polecenie:
`> ipcrm -q ID`
gdzie `ID` jest identyfikatorem usuwanej kolejki komunikatów.

3. Funkcje do operacji na kolejkach komunikatów

a) utworzenie kolejki komunikatów

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

int msgget(key_t key, int msgflg);
```

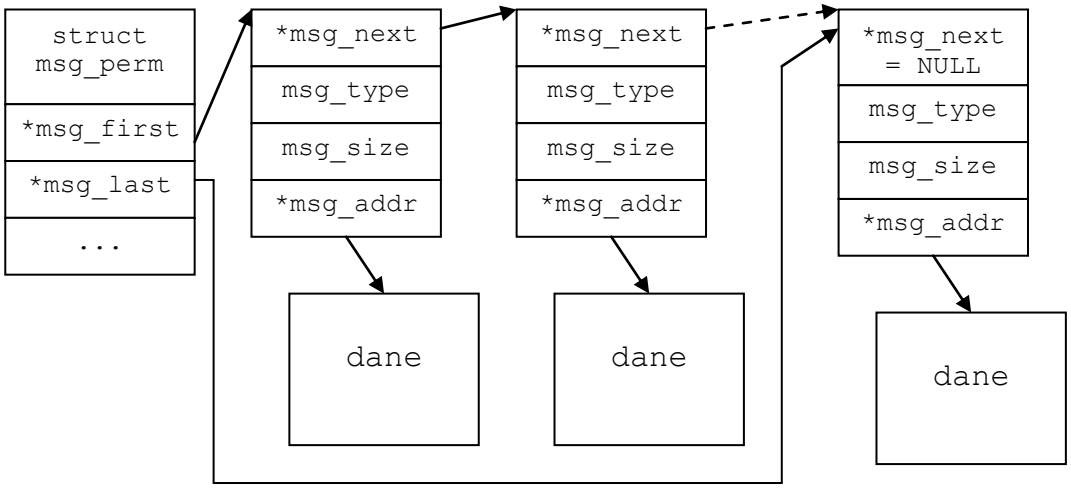
Funkcja `msgget` tworzy w systemie strukturę danych reprezentującą kolejkę komunikatów. Definicja tej struktury znajduje się w pliku nagłówkowym `sys/msg.h`:

```
struct msqid_ds {
   struct ipc_perm   msg_perm;  /* struktura praw dostępu */
   struct msg      *msg_first;  /* wskaźnik do pierwszego komunikatu w
                                   kolejce */
   struct msg       *msg_last;  /* wskaźnik do ostatniego komunikatu w
                                   kolejce */
   msglen_t        msg_cbytes;  /* aktualna liczba bajtów w kolejce */
   msgqnum_t         msg_qnum;  /* aktualna liczba komunikatów w kolejce */
   msglen_t        msg_qbytes;  /* maksymalna dopuszczalna liczba bajtów w
                                   kolejce */
   pid_t            msg_lspid;  /* PID procesu, który ostatnio wysłał
                                   komunikat do kolejki (wywołał msgsnd) */
   pid_t            msg_lrpid;  /* PID procesu, który ostatnio odczytał
                                   komunikat z kolejki (wywołał msgrcv) */
   time_t           msg_stime;  /* czas ostatniego wysłania komuniaktu do
                                   kolejki (msgsnd) */
   time_t           msg_rtime;  /* czas ostatniego odczytu komunikatu z
                                   kolejki (msgrcv) */
   time_t           msg_ctime;  /* czas utworzenia kolejki */
};
```

Powyższa struktura danych zawiera informacje o kolejce komunikatów. Poszczególne komunikaty są przechowywane w strukturach danych typu `msg` powiązanych w listę. Definicja struktury `msg` jest następująca:

```
struct msg {
   struct msg   *msg_next;    /* wskaźnik do następnego komunikatu w
                                 kolejce*/
   long          msg_type;    /* typ komunikatu */
   size_t        msg_size;    /* rozmiar treści komunikatu (bloku danych)*/
   void         *msg_addr;    /* wskaźnik do bloku danych przechowującego
                                 treść komunikatu*/
};
```

Strukturę kolejki komunikatów w jądrze przedstawia poniższy rysunek. Ostatni komunikat w kolejce ma wskaźnik do następnego komunikatu `*msg_next` ustawiony na `NULL`.



Funkcja `msgget` zwraca identyfikator kolejki komunikatów. Identyfikator kolejki jest tworzony na podstawie podanego klucza. Zasady stosowania unikalnego klucza lub flagi `IPC_PRIVATE` są takie same jak dla innych narzędzi IPC. Drugi argument funkcji `msgget` określa sposób tworzenia kolejki i prawa dostępu do kolejki. Podobnie jak dla innych narzędzi IPC można tu stosować flagi `IPC_CREAT` oraz `IPC_EXCL`. Prawa dostępu definiuje się albo za pomocą liczby (w kodzie ósemkowym) albo za pomocą odpowiednich stałych symbolicznych.

Stała symboliczna	**wartość**	**znaczenie**
`MSG_R`	`0400`	prawo czytania dla użytkownika
`MSG_W`	`0200`	prawo zapisu dla użytkownika
`MSG_R >> 3`	`0040`	prawo czytania dla grupy
`MSG_W >> 3`	`0020`	prawo zapisu dla grupy
`MSG_R >> 6`	`0004`	prawo czytania dla pozostałych
`MSG_W >> 6`	`0002`	prawo zapisu dla pozostałych

W przypadku błędu funkcja `msgget` zwraca wartość `-1` i wpisuje kod błędu do globalnej zmiennej `errno`.

b) wysłanie komunikatu

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

int msgsnd(int msqid, const void *msgp,  size_t  msgsz,  int msgflg);
```

Do wysyłania komunikatów służy funkcja `msgsnd`. Pierwszy argument określa identyfikator kolejki komunikatów, do której wysyłany jest komunikat. Drugi argument określa adres, pod którym znajduje się struktura (reprezentująca cały komunikat). Pierwszym elementem tej struktury **musi** być typ komunikatu (liczba typu `long`). Typ komunikatu powinien być liczbą większą od zera. Pozostałe elementy tej struktury, zawierające treść komunikatu (dane) **mogą być dowolne** (definiowane przez programistę). W pliku nagłówkowym `sys/msg.h` zdefiniowana jest struktura komunikatu następującej postaci:

```
struct msgbuf {
        long    mtype;          /* message type */
        char    mtext[1];       /* message text */
};
```

Użytkownik jednak może definiować dowolną inną strukturę, jak to przedstawiono w przykładzie 20. Istotne jest jednak, aby komunikujące się procesy miały uzgodniony format komunikatu. Wskaźnik do zmiennej przechowującej komunikat, przykazywany jako drugi argument `*msgp` powinien być skonwertowany do typu `void *`.

Trzeci argument określa rozmiar danych w bajtach. Argument `msgsz` może mieć wartość `0`. Oznacza to, że przesyłany jest tylko typ komunikatu (struktura komunikatu zawiera jedynie typ).

Jako ostatni argument `msgflg` może wystąpić stała symboliczna `IPC_NOWAIT` lub `0`. Jeżeli `msgflg` ma wartość `0` to w przypadku braku możliwości wysłania komunikatu (np. przepełniona kolejka lub za dużo komunikatów w systemie) proces zawiesi swoje działanie. Ustawienie flagi `IPC_NOWAIT` zapewnia natychmiastowy powrót z funkcji `msgsnd`, która w tym przypadku zwraca wartość `-1` a do zmiennej `errno` zapisywany jest kod błędu `EAGAIN` (brak możliwości wpisania komunikatu do kolejki z powodu przepełnienia lub zbyt dużej liczby komunikatów w systemie).

W przypadku pomyślnego wstawienia komunikatu do kolejki, funkcja zwraca wartość `0`, w przeciwnym przypadku `-1`, a do zmiennej globalnej `errno` jest wstawiany kod błędu.

Przykład 22

Poniżej przedstawiono przykładowe struktury komunikatu definiowane przez programistę. Najczęściej komunikat występuje w postaci ciągu znaków. Struktura ma wówczas następującą postać:

```
typedef struct komunikat {
    long  mtype;
    char  tekst[ROZMIAR]; /* ROZMIAR - długość ciągu znaków */
};
```

Można też definiować dowolną strukturę. Na przykład:

```
typedef struct komunikacik {
    long  mtype;
    int   numer;
    char  tekst[12];
    long  liczba;
    char  znak;
};
```

c) odbieranie komunikatu

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

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int
msgflg);
```

Funkcja `msgrcv` umożliwia pobranie komunikatu z kolejki. Pierwszy argument określa identyfikator kolejki komunikatów, z której pobierany jest komunikat.

Drugi argument `*msgp` określa adres, pod którym znajduje się struktura reprezentująca cały komunikat, do której ma zostać przekopiowany komunikat. Format wskazywanej struktury powinien być zgodny z formatem przesyłanych komunikatów.

Trzeci argument określa rozmiar porcji danych zapisywanych w strukturze wskazywanej przez `*msgp`. Jeżeli wartość argumentu `msgsz` jest mniejsza niż długość odebranego komunikatu wówczas funkcja zgłosi błąd. Ustawienie w tej sytuacji flagi `MSG_NOERROR` (argument `msgflg`) spowoduje, że nie zostanie zgłoszony błąd, nadmiar danych zostanie pominięty, a reszta zostanie odczytana i zapisana w zmiennej wskazywanej przez `*msgp`.

Czwarty argument `msgtyp` określa typ komunikatu, jaki ma być odebrany:

- jeżeli ma on wartość zero, zostanie odebrany pierwszy komunikat z kolejki (niezależnie jakiego jest typu), kolejka jest obsługiwana wg zasady *pierwszy przyszedł – pierwszy wychodzi*
- jeżeli ma wartość dodatnią, zostanie odebrany pierwszy komunikat w kolejce o takim samym typie jak wartość `msgtyp`.
- jeżeli ma wartość ujemną, to odebrany będzie pierwszy komunikat, którego typ jest najmniejszy spośród typów mniejszych lub takich samych, jak wartość bezwzględna typu określonego argumentem `msgtyp`.

Odczytany komunikat jest usuwany z kolejki (nie może być odczytany ponownie, a kolejka przesuwa się do przodu)

Argument `msgflg` określa zachowanie procesu w przypadku, gdy nie można odczytać komunikatu (kolejka jest pusta lub nie ma w niej komunikatu żądanego typu). Jeżeli ustawiono flagę `IPC_NOWAIT` wówczas następuje natychmiastowy powrót z funkcji `msgrcv`, a w zmiennej `errno` zapisywany jest kod błędu `ENOMSG`. Jeżeli `msgflg` ma wartość `0` wówczas proces zawiesza swoje działanie w funkcji `msgrcv` aż do momentu wystąpienia jednego z poniższych zdarzeń:

- w kolejce pojawi się komunikat żądanego typu,
- kolejka komunikatów zostanie usunięta z systemu,
- proces otrzyma sygnał.

Argument `msgflg` może również mieć wartość `MSG_NOERROR`, której zastosowanie opisano wcześniej.

Przypadku odebrania komunikatu funkcja `msgrcv` zwraca liczbę odebranych bajtów porcji danych. W przypadku błędu zwraca `-1`, a do zmiennej `errno` zapisywana jest odpowiednia wartość odpowiadająca błędowi.

d) operacje kontrolne

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Funkcja `msgctl` umożliwia realizację operacji kontrolnych na kolejkach komunikatów. Dzięki niej można między innymi **usunąć z systemu kolejkę komunikatów**. Można również zmieniać prawa dostępu do kolejki. Argument `msqid` określa identyfikator kolejki komunikatów. Drugi argument określa jaka operacja ma być wykonana, a trzeci wskazuje na zmienną typu struktura `msqid_ds`. Możliwe są następujące operacje:

- IPC_STAT – pobiera komplet informacji o kolejce komunikatów i wpisuje ją do struktury wskazywanej przez argument `*buf`,
- IPC_SET – pozwala ustawiać następujące atrybuty kolejki komunikatów:
  - identyfikator właściciela kolejki `msg_perm.uid`
  - identyfikator grupy właściciela kolejki `msg_perm.gid`
  - prawa dostępu do kolejki `msg_perm.mode`
  - maksymalną dopuszczalną liczbę bajtów w kolejce `msg_qbytes`

Można więc zmienić identyfikatory właściciela kolejki. Nie można jednak zmieniać identyfikatorów twórcy kolejki.

- IPC_RMID – **usuwa kolejkę komunikatów** z systemu i likwiduje strukturę danych opisujących ją. Kolejkę może usunąć proces, który ją utworzył, albo inny proces, który ma do tego uprawnienia.

W przypadku błędu funkcja zwraca wartość `-1` i wpisuje kod błędu do globalnej zmiennej `errno`.

Przykład 23

Załóżmy, że w kolejce znajdują się komunikaty następujących typów

pozycja w kolejce:	1	2	3	4	5	6	7	8	9	10	11
typ komunikatu:	2	1	1	5	3	8	2	2	1		
treść:	AA	BB	CC	DD	EE	FF	GG	HH	II		

Procesy zażądały kolejno odczytu komunikatów następujących typów: 1, 5, 1, 2, 0, 0, 2
Kolejno odczytane zostaną następujące komunikaty (typ-treść):

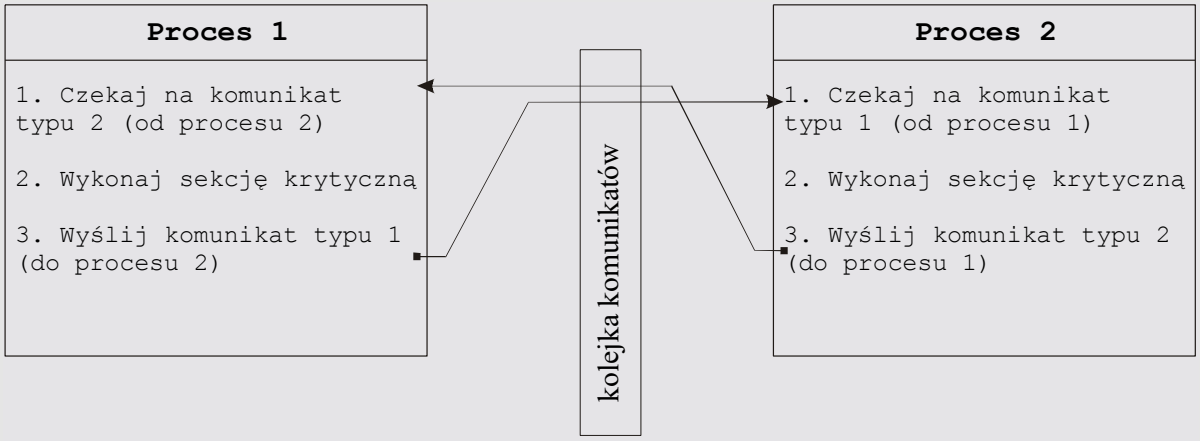
1-BB, 5-DD, 1-CC, 2-AA, 3-EE, 8-FF, 7-GG

4. Przykładowe zastosowania

Na poniższych przykładach pokazano zastosowanie komunikatów do synchronizacji procesów (przykład 3) oraz do tworzenia aplikacji typu klient- server.

Przykład 24

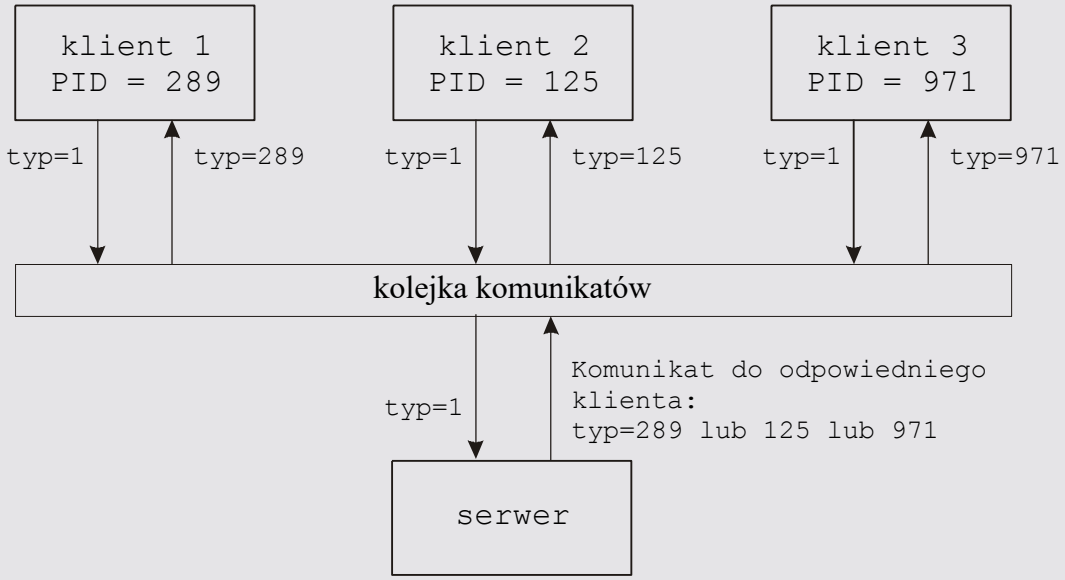
Poniżej przedstawiono przykładowy schemat dwóch współpracujących programów, w których należy zapewnić, aby w danym momencie tylko jeden z nich miał dostęp do sekcji krytycznej.



Proces 1 musi aby wejść do swojej sekcji krytycznej musi czekać na zezwolenie od procesu 2 (oczekuje na komunikat typu 2). Proces 2 po skończeniu swojej sekcji krytycznej wysyła komunikat typu 2 (zezwolenie dla procesu 1 na wejście do sekcji krytycznej). Analogicznie, proces 2 aby wejść do swojej sekcji krytycznej musi czekać na zezwolenie od procesu 1 (komunikat typu 1).

Przykład 25

Na poniższym rysunku przedstawiono schemat działania przykładowej aplikacji klient–serwer.



W tego typu aplikacji istnieje konieczność jednoczesnego przesyłania komunikatów pomiędzy wieloma procesami. Aby rozróżnić komunikaty kierowane do poszczególnych procesów ustalono, że każdy proces oczekuje tylko na jeden ściśle określony typ komunikatu. Dla ułatwienia przyjęto, że każdy proces klienta odbiera komunikaty o typie zgodnym ze swoim identyfikatorem PID. Komunikaty kierowane do serwera mają typ równy 1.

Na podobnej zasadzie można zbudować aplikację, w której wiele procesów będzie komunikować się między sobą (każdy z każdym). Wysyłając komunikat nadaje się mu taki typ jaki jest PID procesu, do którego adresowany jest komunikat. Można dodatkowo w treści komunikatu przesyłać PID procesu wysyłającego.

5. Pytania kontrolne

1. Co to jest kolejka komunikatów?
2. Jakie są ograniczenia na strukturę komunikatu (jakie elementy musi/nie musi ona zawierać)?
3. Czy wysyłając komunikat możemy określić adresata komunikatu?
4. Kto może odebrać komunikat umieszczony w kolejce?
5. Kiedy komunikaty są usuwane z kolejki?
6. W jaki sposób usunąć kolejkę komunikatów z systemu?
7. Jakie wartości może przyjmować typ komunikatu?
8. Co oznacza próba odczytu komunikatu o typie 0?
9. Co się stanie, gdy komunikatu żądanego typu nie ma w kolejce?
10. Jak sprawdzić bieżący rozmiar kolejki komunikatów?

6. Ćwiczenia

Ćwiczenia wykonywane są na serwerze `pluton.kt.agh.edu.pl`
Programy do zajęć znajdują się w katalogu `~rstankie/stud/LAB12`

Uwaga !! Po skończonych zajęciach proszę usunąć pozostałe kolejki komunikatów!!

Ćwiczenie 1:

Skopiuj i skompiluj programy: `m_s.c`, `m_r.c`, `m_rl.c`, `m_rln.c`. Program `m_s` służy do wysyłania komunikatów do kolejki. Komunikat składa się z typu oraz tekstu o długości max. 300 znaków. Typ i treść komunikatu są wprowadzane przez użytkownika. Treść komunikatów może być dowolnym ciągiem znaków. Podanie jako treści komunikatu znaku `q` kończy działanie programu i usuwana jest kolejka komunikatów.

Programy `m_r`, `m_rl` oraz `m_rln` służą do odbierania komunikatów. Program `m_r` odbiera komunikaty o długości 300 bajtów (pełna długość bufora). W programach `m_rl` i `m_rln` długość odbieranego komunikatu jest ograniczona do 2 bajtów. W programie `m_rl` w funkcji `msgrcv` nie ustawiono flagi `MSG_NOERROR`. Jeżeli więc w kolejce znajdzie się komunikat dłuższy, wówczas funkcja zgłosi błąd i program się zakończy. W programie `m_rln` ustawiono tę flagę. Wówczas również komunikaty dłuższe niż 2 bajty będą odbierane ale będą obcinane do 2 bajtów.

Dla realizacji tego ćwiczenia wygodnie będzie uruchomić trzy terminale: jeden na program wysyłający, drugi na program odbierający, trzeci w celu oglądania parametrów kolejki komunikatów.

a)

- Na jednej terminalu uruchom program wysyłający komunikaty `m_s`.
- Zobacz parametry kolejki komunikatów. Czasy `STIME` i `RTIME` powinny być nie ustawione, `LSPID` i `LRPID` równe `0`, liczba komunikatów w kolejce `0`, bieżący rozmiar kolejki `0B`.
- Wyślij komunikaty różnych typów (np.: 2, 5, 6, 2, 7, 5, 5) o dowolnej treści. Należy zwrócić uwagę w jakiej kolejności były umieszczane komunikaty w kolejce. Sprawdź parametry kolejki. Czy `CBYTES` jest równy sumie długości wpisanych komunikatów? Czy liczba komunikatów w kolejce zgadza się z liczbą komunikatów wysłanych?
- Na drugim terminalu uruchom program odbierający `m_r`. Spróbuj odebrać komunikaty różnych typów, w tym 0 (np.: 2, 0, 7, 0, 5, 5). Zaobserwuj w jakiej kolejności komunikaty są wciągane z kolejki. Zażądaj odebrania komunikatu o typie, który nie został wcześniej umieszczony w kolejce. Co się stało? Sprawdź parametry kolejki. Na drugiej pozycji w polu `MODE` powinna pojawić się litera `R`, co oznacza, że jakiś proces zawiesił działanie w oczekiwaniu na odczyt komunikatu.
- Zakończ działanie programów.

b)

Na jednym z terminali uruchom program `m_rl`. Na drugim program `m_s`. Wyślij najpierw kilka komunikatów o treści od długości 1 lub 2 znaków. Następnie wyslij komunikat dłuższy. Przy próbie odebrania komunikatu dłuższego niż 2 bajty program `m_rl` zakończy się z błędem.

c)

Na jednym z terminali uruchomić program `m_rln`. Na drugim program `m_s`. Wysłać komunikaty różnej długości. Program `m_rln` będzie odbierał wszystkie te komunikaty obcinając je jednak do 2 bajtów.

d)

Jeżeli istnieje utworzona wcześniej przez Ciebie kolejka komunikatów, usuń ją. Skopiuj dodatkowo program `msmall.c`. Za jego pomocą utwórz kolejkę komunikatów. Sprawdź jaki jest maksymalny rozmiar tej kolejki w bajtach. Za pomocą programu `m_s` wyślij kilka komunikatów. W pewnym momencie kolejka zostanie przepełniona i program `m_s` zawiesi swoje działanie nie mogąc wysłać komunikatu.

Ćwiczenie 2:

Skopiuj i skompiluj programy: `m_ss.c`, `m_rs.c`, Działają one analogicznie do programów `m_s.c` i `m_r.c`. Różnią się jedynie formatem komunikatu. Przeanalizuj kod, porównaj działanie.

Ćwiczenie 3:

Za pomocą komunikatów zsynchronizuj działanie programów `count1.c` i `count2.c` z zajęć 10. Oczekiwany efekt i wymagania jak w zadaniu 4 z zajęć 10.