Warsztaty HPC: Narzędzia i techniki obliczeń równoległych

Michał Hermanowicz
m.hermanowicz@icm.edu.pl

20-21 stycznia 2025

EuroHPC
Joint Undertaking

Rzeczpospolita
Polska

Narodowe Centrum
Badań i Rozwoju
Unia Europejska
Europejski Fundusz
Rozwoju Regionalnego

Spis treści

1. Wprowadzenie
   - System Unix (AT&T Bell Labs)
   - Projekt GNU, Linux i POSIX
   - Dystrybucje: Przegląd i specyfika
2. Interfejs użytkownika i praca w powłoce
   - GNU Bash
   - Struktura systemu plików
   - Sesja powłoki i operacje na plikach
   - Strumienie I/O i przekierowania
   - Edytor tekstu: `vim`
   - Zmienne i konfiguracja środowiska
   - *Pipeline* – mechanizm potoku
3. Programowanie w powłoce
   - Języki kompilowane vs. interpretowane
   - Skrypty, automatyzacja zadań
4. Użytkownicy i system uprawnień
   - Konta i grupy
   - Pętle, instrukcje warunkowe, arytmetyka
5. Przetwarzanie równoległe
   - Klastry i superkomputery
   - Pamięć współdzielona i rozproszona
   - Slurm: System kolejkowy

Wprowadzenie

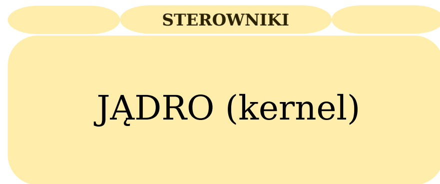
Materiały szkoleniowe i źródła informacji:

- Prezentacja w formacie PDF
- Materiały dodatkowe – instrukcje, przykłady, zadania
- Inne źródła informacji
  - Wybrane strony podręcznika systemowego (`man`)
  - Dokumentacja systemu Debian GNU/Linux (`https://www.debian.org`)
  - Mendel Cooper, *Advanced Bash-Scripting Guide* (`https://tldp.org/LDP/abs/html/`)
  - Zasoby (dokumentacja) omawianych projektów i narzędzi

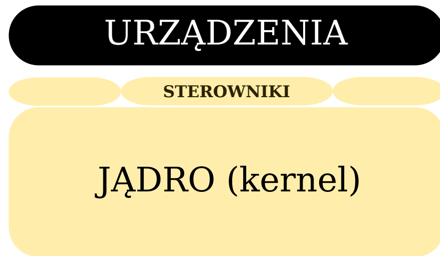
Komponenty systemu operacyjnego

JĄDRO (kernel)

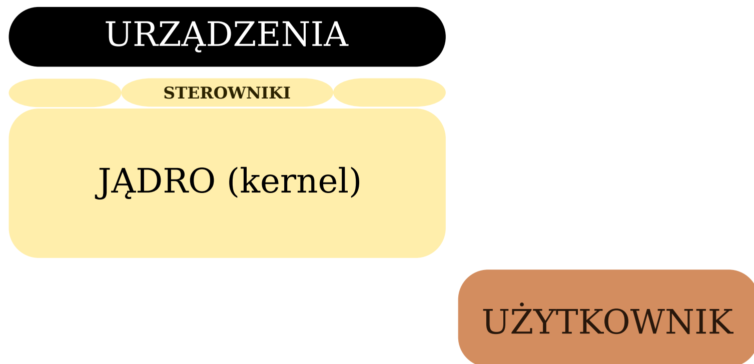
Komponenty systemu operacyjnego



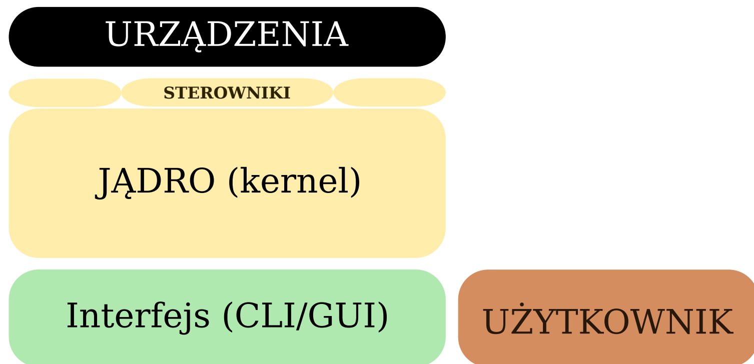
Komponenty systemu operacyjnego



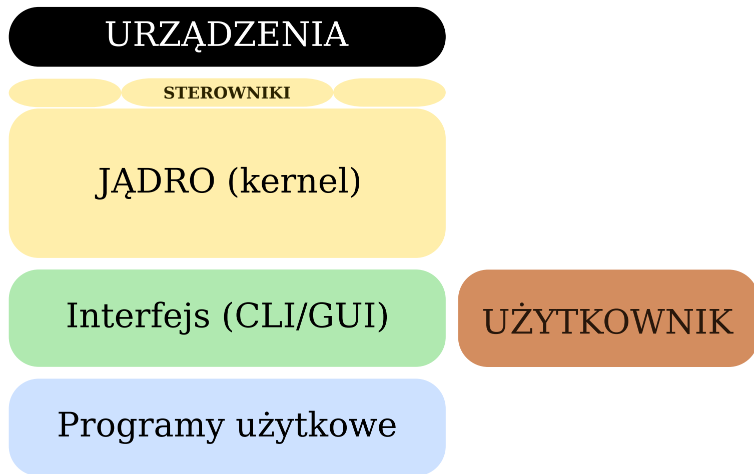
Komponenty systemu operacyjnego



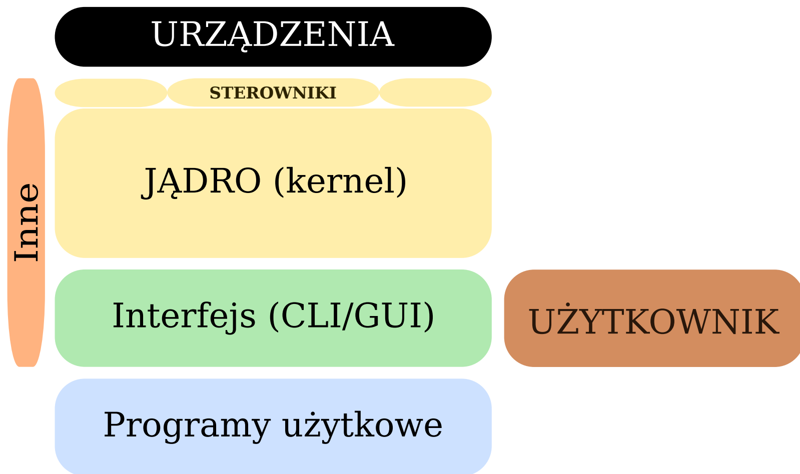
Komponenty systemu operacyjnego



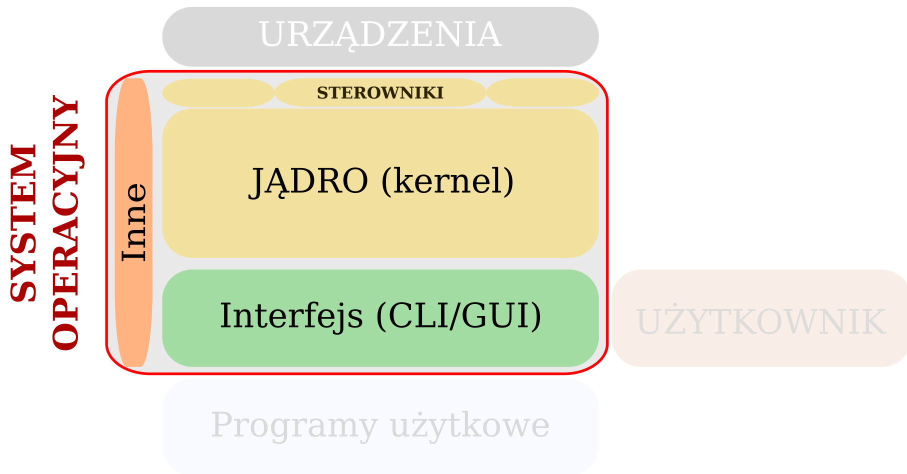
Komponenty systemu operacyjnego



Komponenty systemu operacyjnego



Komponenty systemu operacyjnego



Ken Thompson i Dennis Ritchie (AT&T Bell Labs)

Zdjęcie: **Peter Hamer** [CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0)], via Wikimedia Commons

System operacyjny Unix: filozofia i cechy

Filozofia (autor: Doug McIlroy)[1]:

- *write programs that do **one thing** and do it **well***,
- *write programs to **work together***,
- *write programs that handle **text** streams as **a universal interface***.

---

[1] Cytowane punkty pochodzą z: **Peter H. Salus**, *A Quarter-Century of Unix*. Addison-Wesley, 1994. ISBN 0-201-54777-5.

System operacyjny Unix: filozofia i cechy

Filozofia (autor: Doug McIlroy)[1]:

- *write programs that do* ***one thing*** *and do it* ***well***,
- *write programs to* ***work together***,
- *write programs that handle* ***text*** *streams as* ***a universal interface***.

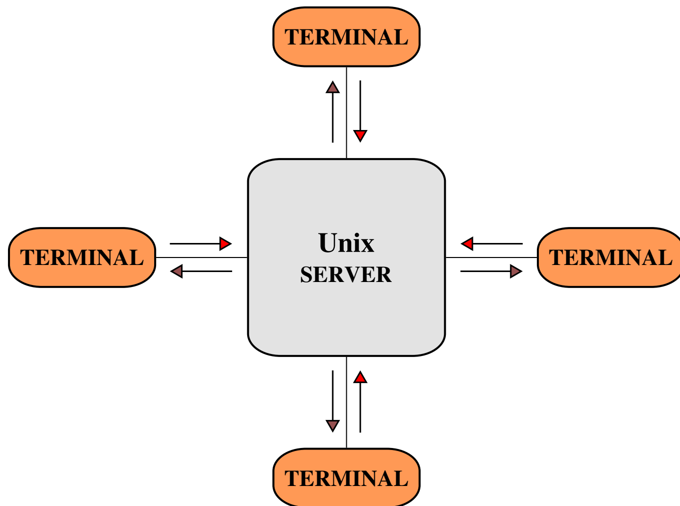
---

[1] Cytowane punkty pochodzą z: **Peter H. Salus**, *A Quarter-Century of Unix*. Addison-Wesley, 1994. ISBN 0-201-54777-5.

Cechy systemu Unix:

- wielozadaniowy, wieloużytkowy system operacyjny (*timesharing* OS),
- monolityczne jądro, hierarchiczny system plików,
- *wszystko jest plikiem* (nawet urządzenia!),
- obsługa sieci, napisany (przepisany) w języku wysokiego poziomu,
- koncepcja ***pipeline*** (*potok*).

System operacyjny Unix: środowisko wieloużytkowe



System operacyjny Unix: terminal **VT100**

Zdjęcie: **Jason Scott** [CC BY 2.0 (http://creativecommons.org/licenses/by/2.0)], via Wikimedia Commons

Projekt GNU

Projekt GNU

Jądro (ang. *kernel*):

- rdzeń systemu operacyjnego,

Projekt GNU

Jądro (ang. *kernel*):

- rdzeń systemu operacyjnego,
- kontroluje dostęp do zasobów sprzętowych i ich wykorzystanie,

Projekt GNU

Jądro (ang. *kernel*):

- rdzeń systemu operacyjnego,
- kontroluje dostęp do zasobów sprzętowych i ich wykorzystanie,
- ładowane do pamięci operacyjnej (ang. *kernel space*) po programie rozruchowym,

Projekt GNU

Jądro (ang. *kernel*):

- rdzeń systemu operacyjnego,
- kontroluje dostęp do zasobów sprzętowych i ich wykorzystanie,
- ładowane do pamięci operacyjnej (ang. *kernel space*) po programie rozruchowym,
- umożliwia pracę innym programom i alokuje dla nich zasoby.

Projekt GNU

Jądro (ang. *kernel*):

- rdzeń systemu operacyjnego,
- kontroluje dostęp do zasobów sprzętowych i ich wykorzystanie,
- ładowane do pamięci operacyjnej (ang. *kernel space*) po programie rozruchowym,
- umożliwia pracę innym programom i alokuje dla nich zasoby.

- Projekt jądra **Hurd**

Projekt GNU: jądro Hurd

GNU Hurd

Recent Changes Preferences *This page:* Edit History Source ?Discussion

What is the GNU Hurd?

The GNU Hurd is the GNU project's replacement for the Unix kernel. It is a collection of servers that run on the Mach microkernel to implement file systems, network protocols, file access control, and other features that are implemented by the Unix kernel or similar kernels (such as Linux). *More detailed*.

What is the mission of the GNU Hurd project?

Our mission is to create a general-purpose kernel suitable for the GNU operating system, which is viable for everyday use, and gives users and programs as much control over their computing environment as possible. *Our mission explained*.

Screenshot: strona domowa projektu GNU Hurd (https://www.gnu.org/software/hurd).

Linux – 1991

Date: 25 August 1991
From: torvalds@kruuna.helsinki.fi

Hello everybody out there using minix --

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things). (...)

Linus (torvalds@kruuna.helsinki.fi)

System operacyjny GNU/Linux

System operacyjny GNU/Linux

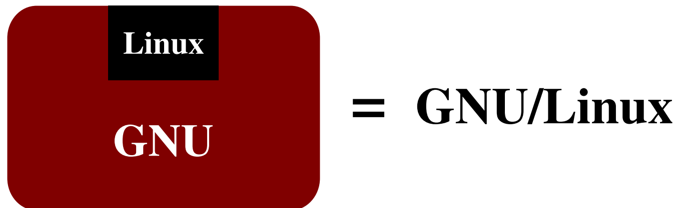
= GNU/Linux

System operacyjny GNU/Linux

= **GNU/Linux**

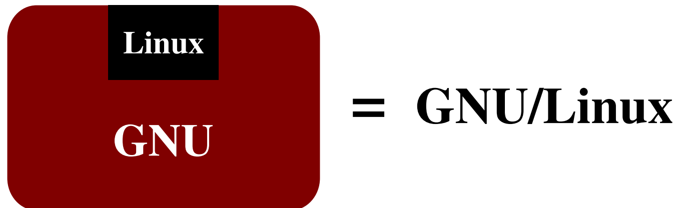
- GNU/Linux to nie jest jeden system operacyjny – istnieje wiele jego **dystrybucji**, z których każda posiada swoją specyfikę.

System operacyjny GNU/Linux



- GNU/Linux to nie jest jeden system operacyjny – istnieje wiele jego **dystrybucji**, z których każda posiada swoją specyfikę.
- Dostępny dla wielu architektur sprzętowych: amd64, arm64, i386, powerpc, mips i in.

System operacyjny GNU/Linux



- GNU/Linux to nie jest jeden system operacyjny – istnieje wiele jego **dystrybucji**, z których każda posiada swoją specyfikę.
- Dostępny dla wielu architektur sprzętowych: amd64, arm64, i386, powerpc, mips i in.
- GNU: www.gnu.org / Linux: www.kernel.org

Dystrybucje: Przegląd i specyfika

Dystrybucja GNU/Linuksa:

to **system operacyjny** złożony ze zbioru programów obejmujących jądro (np. Linux), program rozruchowy (np. GRUB), system zarządzania oprogramowaniem (menedżer pakietów), narzędzia systemowe i programy użytkowe (edytory tekstu, kompilatory, biblioteki, programy graficzne, przeglądarki internetowe/plików i inne).

Dystrybucje: Przegląd i specyfika

Dystrybucja GNU/Linuksa:

to **system operacyjny** złożony ze zbioru programów obejmujących jądro (np. Linux), program rozruchowy (np. GRUB), system zarządzania oprogramowaniem (menedżer pakietów), narzędzia systemowe i programy użytkowe (edytory tekstu, kompilatory, biblioteki, programy graficzne, przeglądarki internetowe/plików i inne).

Który z systemów wybrać?

- Większość to systemy ogólnego przeznaczenia (uniwersalne), choć niektóre z nich mają specyfikę dedykowaną dla konkretnej klasy zastosowań (serwery, komputery osobiste, przenośne, wbudowane).

Dystrybucje: Przegląd i specyfika

Dystrybucja GNU/Linuksa:

to **system operacyjny** złożony ze zbioru programów obejmujących jądro (np. Linux), program rozruchowy (np. GRUB), system zarządzania oprogramowaniem (menedżer pakietów), narzędzia systemowe i programy użytkowe (edytory tekstu, kompilatory, biblioteki, programy graficzne, przeglądarki internetowe/plików i inne).

Który z systemów wybrać?

- Większość to systemy ogólnego przeznaczenia (uniwersalne), choć niektóre z nich mają specyfikę dedykowaną dla konkretnej klasy zastosowań (serwery, komputery osobiste, przenośne, wbudowane).
- Większość zachowuje wspólną filozofię działania i podstawowe zasady standardu POSIX (systemy *posiksowe*).

Dystrybucje: Przegląd i specyfika

- Znajomość standardowych komponentów systemu umożliwia swobodne poruszanie się w środowisku dowolnej dystrybucji na niemal dowolnej platformie sprzętowej.

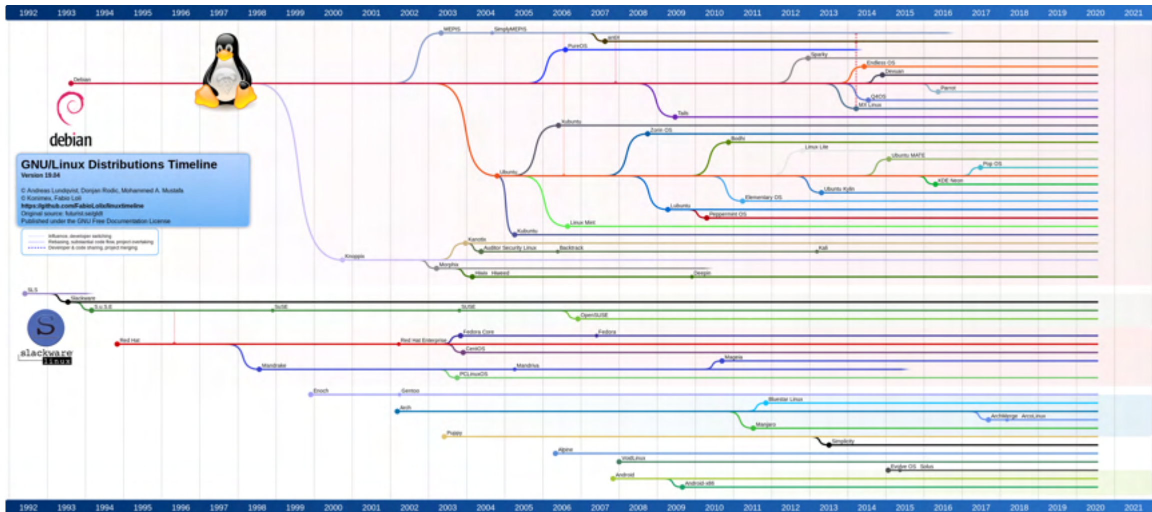
- Znajomość standardowych komponentów systemu umożliwia swobodne poruszanie się w środowisku dowolnej dystrybucji na niemal dowolnej platformie sprzętowej.
- W praktyce różnice sprowadzają się **m.in.** do poziomu automatyzacji czynności systemowych, zarządzania oprogramowaniem i usługami, szczegółów konfiguracji i in.

Dystrybucje: Przegląd i specyfika

- Znajomość standardowych komponentów systemu umożliwia swobodne poruszanie się w środowisku dowolnej dystrybucji na niemal dowolnej platformie sprzętowej.
- W praktyce różnice sprowadzają się **m.in.** do poziomu automatyzacji czynności systemowych, zarządzania oprogramowaniem i usługami, szczegółów konfiguracji i in.

POSIX – *Portable Operating System Interface* – rodzina standardów (IEEE Computer Society) stworzona dla zachowania kompatybilności pomiędzy systemami operacyjnymi. Definiuje API, interfejs użytkownika, narzędzia systemowe i inne.

Dystrybucje: Przegląd i specyfika



System operacyjny GNU/Linux

Niektóre z dystrybucji GNU/Linuksa:

- **Debian** [www.debian.org]
- **Slackware Linux** [www.slackware.com]
- **Arch Linux** [www.archlinux.org]
- **PLD** [www.pld-linux.org]
- **gNewSense** [www.gnewsense.org]
- **Trisquel** [www.trisquel.info]
- **Fedora** [www.getfedora.org]
- **CentOS** [www.centos.org]
- **Scientific Linux** [www.scientificlinux.org]

Logotypy: **GNU** (na licencji GFDL 1.3, źródło: www.gnu.org);
Linux (Tux): Larry Ewing, lewing@isc.tamu.edu, GIMP (www.gimp.org).

Model licencjonowania FLOSS

KOD ŹRÓDŁOWY

```
#include <stdio.h>
int main (void)
{
  puts ("Hello World!");
  return 0;
}
```

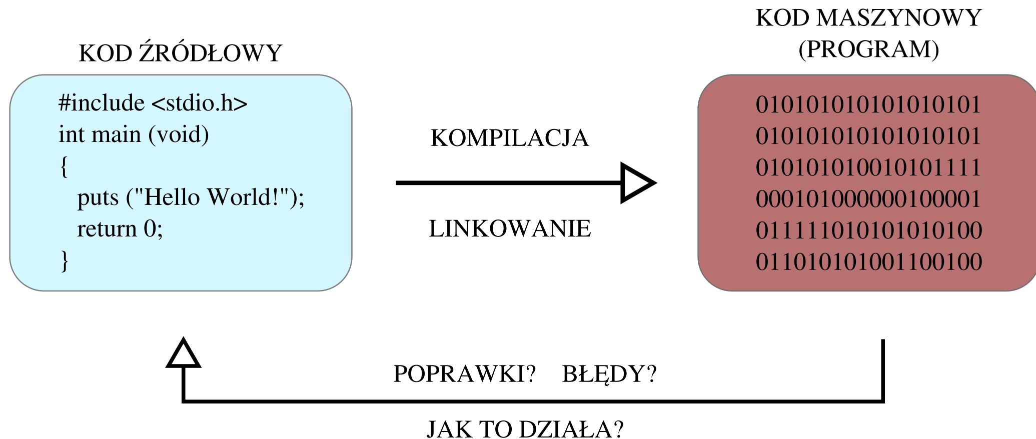
KOMPILACJA

LINKOWANIE

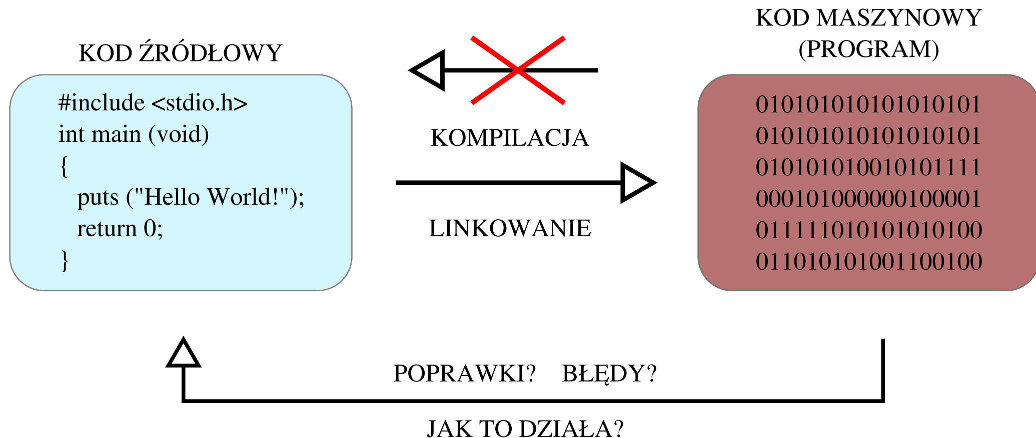
KOD MASZYNOWY (PROGRAM)

010101010101010101
010101010101010101
010101010010101111
000101000000100001
011111010101010100
011010101001100100

Model licencjonowania FLOSS



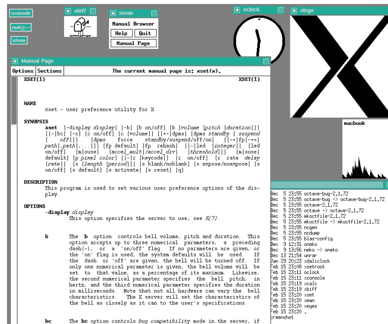
Model licencjonowania FLOSS



System operacyjny GNU/Linux: **interfejs użytkownika**

```
cron.weekly                    hosts
csh                            hosts.allow
csh.cshrc                      hosts.deny
csh.login                      hp
csh.logout                     htdig
cups                           i3
cupshelpers                    i3status.conf
dbus-1                         icedove
debconf.conf                   icedtea-web
debian_version                 iceweasel
default                        idmapd.conf
deluser.conf                   ifplugd
dhcp                           ImageMagick-6
dictionaries-common            init
discover.conf.d                init.d
discover-modprobe.conf         initramfs-tools
dkms                           inputrc
dpkg                           insserv
drirc                          insserv.conf
herman@lepton:/etc$
herman@lepton:/etc$
herman@lepton:/etc$
herman@lepton:/etc$
```

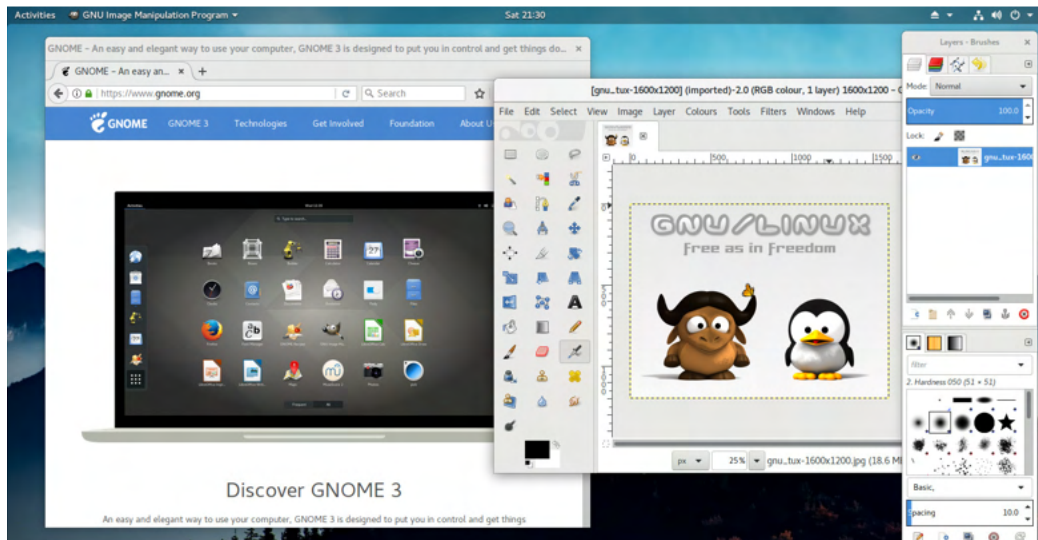
CLI
(Command Line Interface)



GUI
(Graphical User Interface)

Zrzut ekranu z prawej: Liberal Classic [MIT (http://opensource.org/licenses/mit-license.php)], via Wikimedia Commons

System operacyjny GNU/Linux: GUI (GNOME)



GNU Bash (**B**ourne **A**gain **SH**ell):

interpreter języka poleceń – **powłoka** (*shell*) systemu GNU/Linux[1].

---

[1] *The GNU Bash Reference Manual*, v. 4.3 (http://www.gnu.org).

GNU Bash (**B**ourne **A**gain **SH**ell):

interpreter języka poleceń – **powłoka** (*shell*) systemu GNU/Linux[1].

---

[1] *The GNU Bash Reference Manual*, v. 4.3 (http://www.gnu.org).

Ponadto:

- jest domyślną powłoką systemu GNU/Linux – jego **CLI** (Command Line Interface),
- umożliwia pracę interaktywną (wprowadzanie poleceń), a także wsadową (wykonywanie skryptów).

Inne powłoki: `sh`, `csh`, `tcsh`, `ksh`, `zsh`.

- Powłoka jest językiem programowania pozwalającym na łączenie funkcjonalności wielu różnych programów i narzędzi systemowych, a także wewnętrznych elementów samego języka – funkcji, pętli, instrukcji warunkowych i innych.

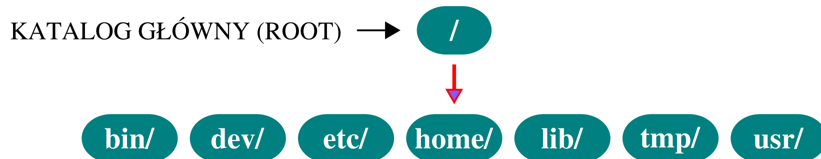
- Powłoka jest językiem programowania pozwalającym na łączenie funkcjonalności wielu różnych programów i narzędzi systemowych, a także wewnętrznych elementów samego języka – funkcji, pętli, instrukcji warunkowych i innych.

- Poza bieżącą obsługą systemu pozwala na wykonywanie powtarzalnych i czasochłonnych zadań administracyjnych oraz automatyzację czynności (jest to język konfiguracji systemu).

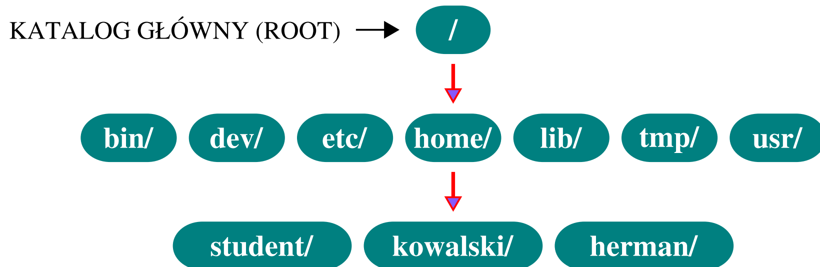
O systemie plików

System operacyjny GNU/Linux: **system plików**

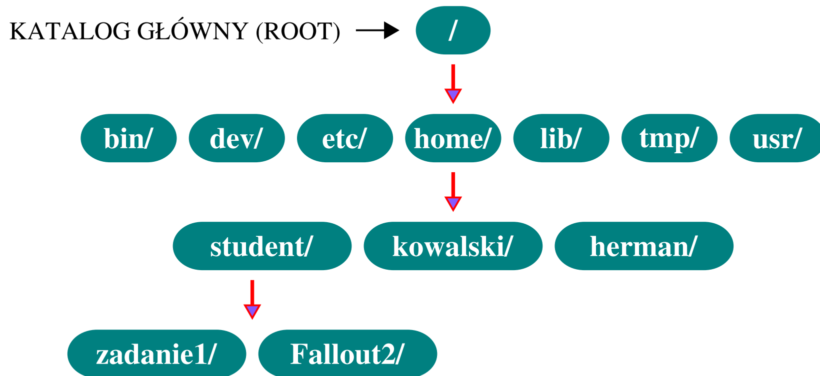
System operacyjny GNU/Linux: **system plików**



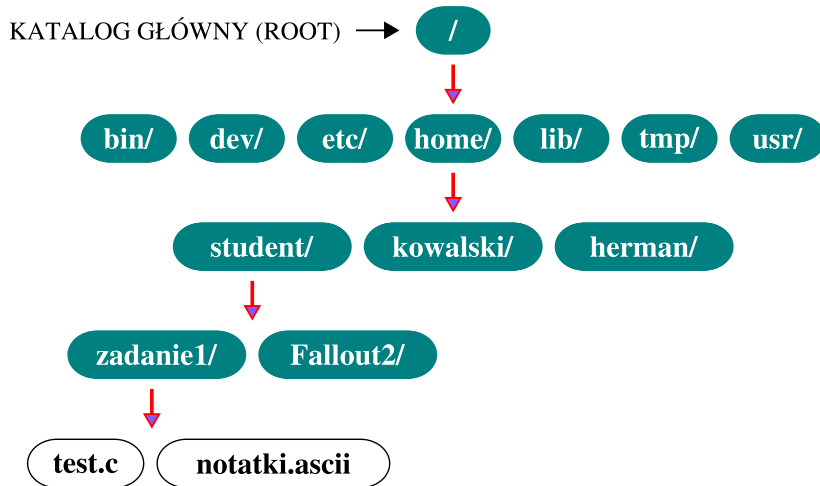
System operacyjny GNU/Linux: **system plików**



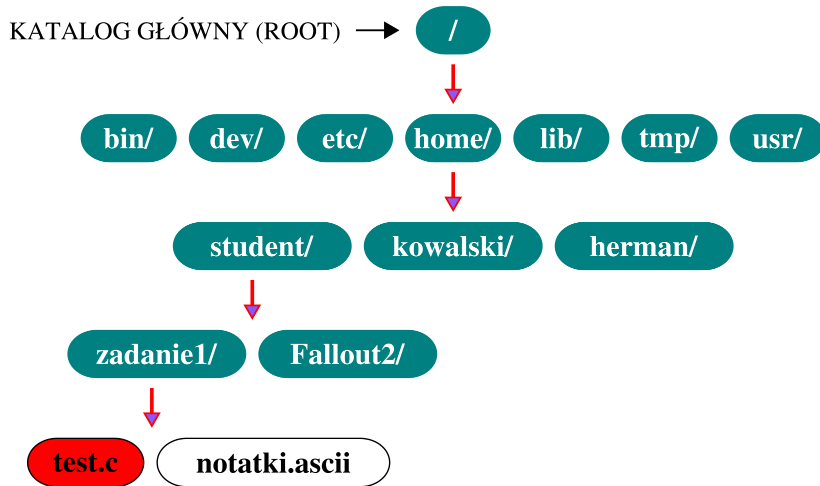
System operacyjny GNU/Linux: **system plików**

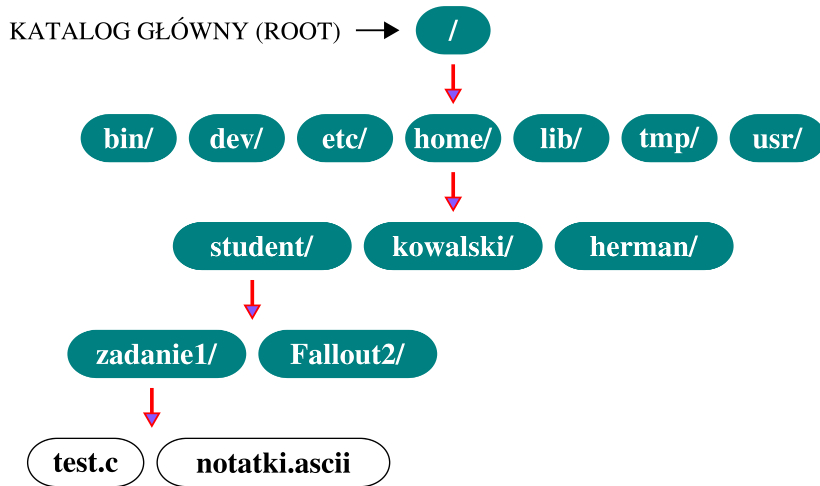


System operacyjny GNU/Linux: **system plików**

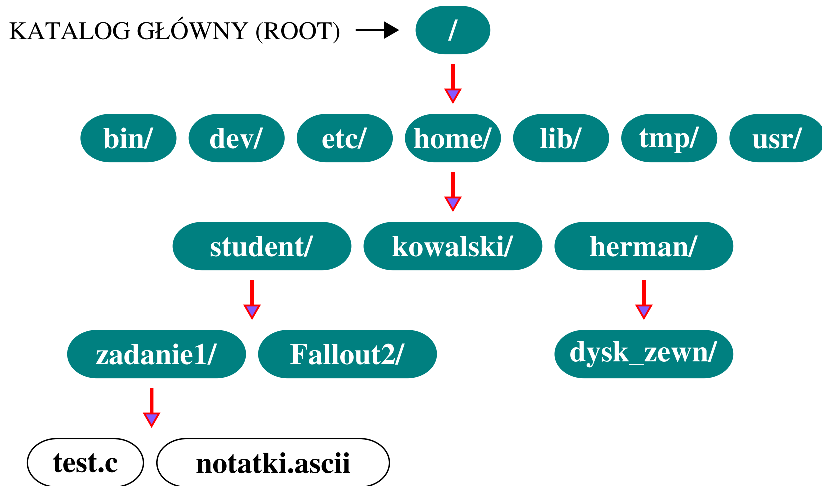


System operacyjny GNU/Linux: **system plików**




KATALOG GŁÓWNY (ROOT)
/
bin/
dev/
etc/
home/
lib/
tmp/
usr/
student/
kowalski/
herman/
zadanie1/
Fallout2/
test.c
notatki.ascii

System operacyjny GNU/Linux: **system plików**



Bezwzględna ścieżka dostępu:

`/home/student/zadanie1/test.c`

System operacyjny GNU/Linux: **system plików**

Bezwzględna ścieżka dostępu:

`/home/student/zadanie1/test.c`

Względna ścieżka dostępu:

`zadanie1/test.c`

System operacyjny GNU/Linux: **system plików**

Bezwzględna ścieżka dostępu:

`/home/student/zadanie1/test.c`

Względna ścieżka dostępu:

`zadanie1/test.c`

Ważne katalogi:

- `/bin` – pliki wykonywalne (***bin**ary*), programy użytkowe i inne,
- `/dev` – pliki urządzeń (dyski, drukarki i inne),
- `/etc` – pliki konfiguracyjne systemu,
- `/home` – katalogi domowe użytkowników,
- `/lib` – biblioteki systemowe,
- `/tmp` – pliki tymczasowe,
- `/usr` – oprogramowanie użytkownika wraz z bibliotekami, dok. i in.

System operacyjny GNU/Linux: **system plików**

W nazwach plików i katalogów:

- **NIE** używamy SPACJI*,
- **NIE** używamy znaków diakrytycznych*.

*Choć jest to technicznie możliwe, to znacznie komplikuje wiele czynności i w efekcie utrudnia korzystanie z systemu, dlatego taką zasadę warto przyjąć jako dobrą praktykę.

System operacyjny GNU/Linux: **system plików**

W nazwach plików i katalogów:

- **NIE** używamy SPACJI*,
- **NIE** używamy znaków diakrytycznych*.

*Choć jest to technicznie możliwe, to znacznie komplikuje wiele czynności i w efekcie utrudnia korzystanie z systemu, dlatego taką zasadę warto przyjąć jako dobrą praktykę.

Domyślnie użytkownik ma uprawnienia do zapisu w katalogach:

- `/home/uzytkownik/`
- `/tmp/`

System operacyjny GNU/Linux: **system plików**

W nazwach plików i katalogów:

- **NIE** używamy SPACJI*,
- **NIE** używamy znaków diakrytycznych*.

*Choć jest to technicznie możliwe, to znacznie komplikuje wiele czynności i w efekcie utrudnia korzystanie z systemu, dlatego taką zasadę warto przyjąć jako dobrą praktykę.

Domyślnie użytkownik ma uprawnienia do zapisu w katalogach:

- `/home/uzytkownik/`
- `/tmp/`

Niektóre z obsługiwanych systemów plików:

EXT2/EXT3/EXT4, ReiserFS, XFS, ZFS, JFS, FAT16/32, NTFS

Sesja powłoki i operacje na plikach

```
student@wftlab-180:~$
```

Uruchamianie programu – składnia:

```
program [OPCJA] [ARGUMENT]
```

```
student@wftlab-180:~$
```

Sesja powłoki i operacje na plikach

```
student@wftlab-180:~$
```

Uruchamianie programu – składnia:

```
program [OPCJA] [ARGUMENT]
```

```
student@wftlab-180:~$ ls -l zadanie1
```

Sesja powłoki i operacje na plikach

```
student@wftlab-180:~$
```

Uruchamianie programu – składnia:

```
program [OPCJA] [ARGUMENT]
```

```
student@wftlab-180:~$ ls -l zadanie1
razem 0
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 notatki.ascii
```

Sesja powłoki i operacje na plikach

```
student@wftlab-180:~$
```

Uruchamianie programu – składnia:

```
program [OPCJA] [ARGUMENT]
```

```
student@wftlab-180:~$ ls -l zadanie1
razem 0
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 notatki.ascii
```

`.x`	plik/katalog **ukryty**
`..`	katalog **nad**rzędny
`.`	katalog **bieżący**
`~`	katalog **domowy**

Powłoka: Konfiguracja środowiska

Plik:	Kiedy jest wykonywany?
~/.bash_profile	Jednokrotnie podczas logowania
~/.bashrc	Podczas logowania oraz przy starcie każdej sesji powłoki
~/.profile	Przy braku ~/.bash_profile i ~/.bash_login
~/.bash_login	Podczas logowania, przy braku ~/.bash_profile
~/.bash_logout	Podczas wylogowania
~/.inputrc	Podczas logowania (definiuje skróty klawiszowe)
/etc/profile	Jednokrotnie podczas logowania (zmienne środowiskowe)
/etc/bashrc	Analogicznie do ~/.bashrc

Powłoka: Konfiguracja środowiska

Plik:	**Kiedy jest wykonywany?**
`~/.bash_profile`	Jednokrotnie podczas logowania
`~/.bashrc`	Podczas logowania oraz przy starcie każdej sesji powłoki
`~/.profile`	Przy braku ~/.bash_profile i ~/.bash_login
`~/.bash_login`	Podczas logowania, przy braku ~/.bash_profile
`~/.bash_logout`	Podczas wylogowania
`~/.inputrc`	Podczas logowania (definiuje skróty klawiszowe)
`/etc/profile`	Jednokrotnie podczas logowania (zmienne środowiskowe)
`/etc/bashrc`	Analogicznie do ~/.bashrc

Historia wprowadzonych instrukcji przechowywana jest w pliku `~/.bash_history` (zapisywany na zakończenie sesji). Przydatne polecenia: `history`, `fc -l`.

Strumienie I/O i przekierowania

Strumienie I/O i przekierowania

Trzy elementarne strumienie danych:

- `STDIN (0)` – STanDard INput – standardowe WEjście,
- `STDOUT (1)` – STanDard OUTput – standardowe WYjście,
- `STDERR (2)` – STanDard ERRor – standardowe wyjście diagnostyczne (błędy).

Strumienie I/O i przekierowania

Trzy elementarne strumienie danych:

- `STDIN (0)` – STanDard INput – standardowe WEjście,
- `STDOUT (1)` – STanDard OUTput – standardowe WYjście,
- `STDERR (2)` – STanDard ERRor – standardowe wyjście diagnostyczne (błędy).

- Każdy plik posiada stowarzyszony z nim deskryptor (ang. *file descriptor*),
- urządzeniem `stdin` jest klawiatura,
- urządzeniem `stdout` i `stderr` jest ekran (ale może nim być plik, a nawet urządzenie peryferyjne!).

Strumienie I/O i przekierowania

Przekierowanie standardowego WEjścia:

```
polecenie < input.txt
```

Strumienie I/O i przekierowania

Przekierowanie standardowego WEjścia:

```
polecenie < input.txt
```

Standardowe wyjście (*stdout*, **1**)

```
$ ./hello.sh
Hello World!
```

← standardowe WYjście

Strumienie I/O i przekierowania

Przekierowanie standardowego WEjścia:

```
polecenie < input.txt
```

Standardowe wyjście (*stdout*, **1**)

```
$ ./hello.sh
Hello World!
```

← standardowe WYjście

Przekierowanie *stdout* do pliku:

```
$ ./hello.sh > plik
```

Strumienie I/O i przekierowania

Przekierowanie standardowego WEjścia:

```
polecenie < input.txt
```

Standardowe wyjście (*stdout*, **1**)

```
$ ./hello.sh
Hello World!
```

← standardowe WYjście

Przekierowanie *stdout* do pliku:

```
$ ./hello.sh > plik
```

Alternatywnie:

```
$ ./hello.sh 1>plik
```

Strumienie I/O i przekierowania

Analogicznie (*stderr* = 2):

```
$ ./hello.sh 2>plik_err
```

Strumienie I/O i przekierowania

Analogicznie (*stderr* = 2):

```
$ ./hello.sh 2>plik_err
```

***stderr* → *stdout*:**

```
$ ./hello.sh 2>&1
```

Strumienie I/O i przekierowania

Analogicznie (*stderr* = 2):

```
$ ./hello.sh 2>plik_err
```

stderr → *stdout*:

```
$ ./hello.sh 2>&1
```

stdout i *stderr* do **jednego** pliku:

```
$ ./hello.sh &>plik
```

Strumienie I/O i przekierowania

Analogicznie (*stderr* = 2):

```
$ ./hello.sh 2>plik_err
```

***stderr* → *stdout*:**

```
$ ./hello.sh 2>&1
```

***stdout* i *stderr* do jednego pliku:**

```
$ ./hello.sh &>plik
```

Można też:

```
$ ./hello.sh &>/dev/null
```

Strumienie I/O i przekierowania

Analogicznie (*stderr* = 2):

```
$ ./hello.sh 2>plik_err
```

***stderr* → *stdout*:**

```
$ ./hello.sh 2>&1
```

***stdout* i *stderr* do jednego pliku:**

```
$ ./hello.sh &>plik
```

Można też:

```
$ ./hello.sh &>/dev/null
```

A także:

```
$ ./hello.sh 1>plik 2>bledy
```

Strumienie I/O i przekierowania

/dev/null

to plik specjalny – urządzenie zerowe (tzw. *czarna dziura*). Przekierowanie informacji do tego pliku powoduje ich utratę (bezpowrotne usunięcie).

Strumienie I/O i przekierowania

/dev/null

to plik specjalny – urządzenie zerowe (tzw. *czarna dziura*). Przekierowanie informacji do tego pliku powoduje ich utratę (bezpowrotne usunięcie).

W systemie GNU/Linux wszystko jest plikiem, również urządzenia, dlatego można zrobić tak:

```
$ ./hello.sh > /dev/lpr
```

`/dev/lpr` to drukarka. Jeżeli jest podłączona do komputera i skonfigurowana w systemie – wówczas wynik działania skryptu zostanie wydrukowany.

Strumienie I/O i przekierowania

/dev/null

to plik specjalny – urządzenie zerowe (tzw. *czarna dziura*). Przekierowanie informacji do tego pliku powoduje ich utratę (bezpowrotne usunięcie).

W systemie GNU/Linux wszystko jest plikiem, również urządzenia, dlatego można zrobić tak:

```
$ ./hello.sh > /dev/lpr
```

`/dev/lpr` to drukarka. Jeżeli jest podłączona do komputera i skonfigurowana w systemie – wówczas wynik działania skryptu zostanie wydrukowany.

Inne pliki urządzeń: skanery, plotery, dyski twarde, napędy CD-ROM, stacje dyskietek, streamery itd.

Edytor tekstu `vim` (*vi improved*)

```
\section{Programowanie w środowisku GNU/Linux}
\subsection{Wprowadzenie}

\frame{
\frametitle{Języki kompilowane i interpretowane}

\begin{block}{Jeden z możliwych podziałów języków programowania pozwala wyodrębnić dwie (niejednoznaczne) kategorie:}
\begin{itemize}
\item języki \textbf{kompilowane} -- kod źródłowy jest tłumaczony na kod maszynowy za pomocą kompilatora (ANSI C, C++, FORTRAN i in.),
\item języki \textbf{interpretowane} -- kod źródłowy jest wykonywany bezpośrednio przez interpreter (bash, Python, Perl, PHP i in.).
\end{itemize}
\end{block}

\pause

\begin{exampleblock}{Narzędzia programistyczne:}
\begin{itemize}
\item edytor tekstu, kompilator (w tym linker)/interpreter, debugger,
\item opcjonalnie: środowisko programistyczne zawierające funkcjonalność wszystkich powyższych narzędzi (i więcej).
\end{itemize}
\end{exampleblock}

\small
Przykładowe narzędzia:
\begin{itemize}
\item vim, emacs (edytory),
\item gcc, gfortran (kompilatory); gdb (debugger).
\end{itemize}
}
```

- Edytor tekstu wydany na wiele plaform systemowych; bogactwo możliwości edycyjnych; wygodne *środowisko* programistyczne w połączeniu z kompilatorem gcc i narzędziami powłoki;
- wiele trybów edycji – szczegóły w czasie sesji praktycznej).

Edytor tekstu `vim` (*vi improved*)

```
$ vim nazwa_pliku
```

- `:w nazwa` – zapis do pliku,
- `:wq` – zapis i wyjście,
- `:q!` – wyjście (ignoruj zmiany),
- `i` – tryb edycji,
- `[ESC]` – opuść tryb edycji,
- `u` – cofnij,
- `<Ctrl-R>` – powtórz,
- `/wzorzec` – szukaj wzorca (`n` – nast. / `N` – poprzedni),
- `:!polecenie` – polec. powłoki,
- `dd` – usuń bieżący wiersz,
- `D` – usuń stąd do końca wiersza,
- `:%s/raz/dwa/g` – znajdź i zastąp (*raz*→*dwa*),
- `:s/raz/dwa/g` – znajdź i zastąp w bieżącym wierszu,
- `:s/raz/dwa/gc` – znajdź i zastąp z potwierdzeniem,
- `v` – zaznacz (przesuwając kursor: `[←]`, `[↑]`, `[↓]`, `[→]`),
- `y` – skopiuj zaznaczenie,
- `d` – wytnij zaznaczenie,
- `p` – wklej skopiowane.

Powłoka: Zmienne (środowiskowe)

Zmienne:

W analogii do innych języków programowania – symbole reprezentujące dane (liczba, ciąg znaków), pozwalające na ich przetwarzanie (przechowywanie, odczyt, zapis, manipulacja).

Powłoka: Zmienne (środowiskowe)

Zmienne:

W analogii do innych języków programowania – symbole reprezentujące dane (liczba, ciąg znaków), pozwalające na ich przetwarzanie (przechowywanie, odczyt, zapis, manipulacja).

Tworzenie zmiennych i przypisywanie wartości:

```
var1=ciag-znakow
zmienna=1
a=10
```

Powłoka: Zmienne (środowiskowe)

Zmienne:

W analogii do innych języków programowania – symbole reprezentujące dane (liczba, ciąg znaków), pozwalające na ich przetwarzanie (przechowywanie, odczyt, zapis, manipulacja).

Tworzenie zmiennych i przypisywanie wartości:

```
var1=ciag-znakow
zmienna=1
a=10
```

```
echo $zmienna
1
```

Powłoka: Zmienne (środowiskowe)

Zmienne:

W analogii do innych języków programowania – symbole reprezentujące dane (liczba, ciąg znaków), pozwalające na ich przetwarzanie (przechowywanie, odczyt, zapis, manipulacja).

Tworzenie zmiennych i przypisywanie wartości:

```
var1=ciag-znakow
zmienna=1
a=10
```

```
echo $zmienna
1
```

```
echo $a+1
10+1
```

Powłoka: Zmienne (środowiskowe)

```
$($a+1)
bash: 10+1: nie znaleziono polecenia
```

Powłoka: Zmienne (środowiskowe)

```
$($a+1)
bash: 10+1: nie znaleziono polecenia
```

```
$((a+1))
bash: 11: nie znaleziono polecenia
```

Powłoka: Zmienne (środowiskowe)

```
$($a+1)
bash: 10+1: nie znaleziono polecenia
```

```
$((a+1))
bash: 11: nie znaleziono polecenia
```

```
echo $(($a+1))
11
```

Powłoka: Zmienne (środowiskowe)

```
$($a+1)
bash: 10+1: nie znaleziono polecenia
```

```
$((a+1))
bash: 11: nie znaleziono polecenia
```

```
echo $(($a+1))
11
```

eval:

tworzy (składa) instrukcję ze zmiennych i wykonuje ją.

Powłoka: Zmienne (środowiskowe)

```
$($a+1)
bash: 10+1: nie znaleziono polecenia
```

```
$((a+1))
bash: 11: nie znaleziono polecenia
```

```
echo $(($a+1))
11
```

eval:

tworzy (składa) instrukcję ze zmiennych i wykonuje ją.

```
polecenie=echo
```

Powłoka: Zmienne (środowiskowe)

```
$($a+1)
bash: 10+1: nie znaleziono polecenia
```

```
$((a+1))
bash: 11: nie znaleziono polecenia
```

```
echo $(($a+1))
11
```

eval:

tworzy (składa) instrukcję ze zmiennych i wykonuje ją.

```
polecenie=echo
eval $polecenie $a+1
```

Powłoka: Zmienne (środowiskowe)

Zmienna środowiskowa:

w odróżnieniu od zmiennych lokalnych – jest dziedziczona przez procesy potomne .

Powłoka: Zmienne (środowiskowe)

Zmienna środowiskowa:

w odróżnieniu od zmiennych lokalnych – jest dziedziczona przez procesy potomne .

Zmienne utworzona jako:

```
a=1
```

jest dostępna jedynie dla bieżącej sesji powłoki. Żeby uczynić z niej zmienną globalną (środowiskową) należy użyc instrukcji:

```
export a=1
```

lub chcąc *wyeksportować* zdefiniowaną już zmienną lokalną:

```
export a
```

Powłoka: Zmienne (środowiskowe)

Zmienna środowiskowa:

w odróżnieniu od zmiennych lokalnych – jest dziedziczona przez procesy potomne .

Zmienne utworzona jako:

```
a=1
```

jest dostępna jedynie dla bieżącej sesji powłoki. Żeby uczynić z niej zmienną globalną (środowiskową) należy użyc instrukcji:

```
export a=1
```

lub chcąc *wyeksportować* zdefiniowaną już zmienną lokalną:

```
export a
```

Inne narzędzia o podobnej (lecz nie takiej samej) funkcjonalności: `declare`, `env`.

Powłoka: Zmienne (środowiskowe)

Przykłady zmiennych środowiskowych:

- `$PATH` – przechowuje ścieżki dostępu do programów/skryptów,
- `$PWD` – ścieżka bieżącego katalogu roboczego,
- `$HOME` – ścieżka do katalogu domowego użytkownika,
- `$HOSTNAME` – nazwa hosta,
- `$SHELL` – powłoka użytkownika,
- `$USER` – nazwa użytkownika.

Powłoka: Zmienne (środowiskowe)

Przykłady zmiennych środowiskowych:

- $PATH – przechowuje ścieżki dostępu do programów/skryptów,
- $PWD – ścieżka bieżącego katalogu roboczego,
- $HOME – ścieżka do katalogu domowego użytkownika,
- $HOSTNAME – nazwa hosta,
- $SHELL – powłoka użytkownika,
- $USER – nazwa użytkownika.

```
echo $PATH
```

Powłoka: Zmienne (środowiskowe)

Przykłady zmiennych środowiskowych:

- $PATH – przechowuje ścieżki dostępu do programów/skryptów,
- $PWD – ścieżka bieżącego katalogu roboczego,
- $HOME – ścieżka do katalogu domowego użytkownika,
- $HOSTNAME – nazwa hosta,
- $SHELL – powłoka użytkownika,
- $USER – nazwa użytkownika.

```
echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

```
polecenie1 | polecenie2 | polecenie3 | ... | polecenieN
```

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

```
polecenie1 | polecenie2 | polecenie3 | ... | polecenieN
```

(**Zły**) Przykład z użyciem narzędzia grep:

```
$ cat liczby.txt | grep 23
```

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

```
polecenie1 | polecenie2 | polecenie3 | ... | polecenieN
```

(**Zły**) Przykład z użyciem narzędzia grep:

```
$ cat liczby.txt | grep 23
23
245723
```

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

```
polecenie1 | polecenie2 | polecenie3 | ... | polecenieN
```

(**Zły**) Przykład z użyciem narzędzia grep:

```
$ cat liczby.txt | grep 23
23
245723
```

Narzędzie `wc` (*word count*) - zlicza słowa, wiersze, bajty:

`$ wc -l liczby.txt` (← z przełącznikiem `-l` zlicza wiersze)

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

`polecenie1 | polecenie2 | polecenie3 | ... | polecenieN`

(**Zły**) Przykład z użyciem narzędzia grep:

```
$ cat liczby.txt | grep 23
23
245723
```

Narzędzie `wc` (*word count*) - zlicza słowa, wiersze, bajty:

```
$ wc -l liczby.txt
5
```

(← z przełącznikiem `-l` zlicza wiersze)

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

```
polecenie1 | polecenie2 | polecenie3 | ... | polecenieN
```

(**Zły**) Przykład z użyciem narzędzia grep:

```
$ cat liczby.txt | grep 23
23
245723
```

Narzędzie `wc` (*word count*) - zlicza słowa, wiersze, bajty:

`$ wc -l liczby.txt` (← z przełącznikiem `-l` zlicza wiersze)

```
5
$ cat liczby.txt | grep 23 | wc -l
```

Potok (*pipeline*)

Potok – *pipeline* (symbol: |)

przekierowuje standardowe WYjście (*stdout*) programu na WEjście innego.

```
polecenie1 | polecenie2 | polecenie3 | ... | polecenieN
```

(**Zły**) Przykład z użyciem narzędzia grep:

```
$ cat liczby.txt | grep 23
23
245723
```

Narzędzie wc (*word count*) - zlicza słowa, wiersze, bajty:

$ wc -l liczby.txt (← z przełącznikiem -l zlicza wiersze)

```
5
$ cat liczby.txt | grep 23 | wc -l
2
```

Potok (*pipeline*)

Narzędzie `tee` (`man tee`) czyta WEjście (*stdin*) i zapisuje na WYjście (*stdout*) **oraz** do pliku.

Przykład:

```
$ cat liczby.txt | tee out
```

Potok (*pipeline*)

Narzędzie `tee` (`man tee`) czyta WEjście (*stdin*) i zapisuje na WYjście (*stdout*) **oraz** do pliku.

Przykład:

```
$ cat liczby.txt | tee out
45678
23
2435
67876
245723
```

Potok (*pipeline*)

Narzędzie `tee` (`man tee`) czyta WEjście (*stdin*) i zapisuje na WYjście (*stdout*) **oraz** do pliku.

Przykład:

```
$ cat liczby.txt | tee out
45678
23
2435
67876
245723
$ cat out
```

Potok (*pipeline*)

Narzędzie `tee` (`man tee`) czyta WEjście (*stdin*) i zapisuje na WYjście (*stdout*) **oraz** do pliku.

Przykład:

```
$ cat liczby.txt | tee out
45678
23
2435
67876
245723
$ cat out
45678
23
2435
67876
245723
```

Sesja powłoki i operacje na plikach

`grep` (`man grep`)

przeszukuje zadany plik (lub *stdout*) w poszukiwaniu wierszy zawierających określony *wzorzec*.

SKŁADNIA: `grep WZORZEC plik`

Sesja powłoki i operacje na plikach

grep (man grep)

przeszukuje zadany plik (lub *stdout*) w poszukiwaniu wierszy zawierających określony *wzorzec*.

SKŁADNIA: `grep WZORZEC plik`

Przykład:

```
$ cat liczby.txt
45678
23
2435
67876
245723
$ grep 23 liczby.txt
23
245723
```

Języki kompilowane i interpretowane

Jeden z możliwych podziałów języków programowania pozwala wyodrębnić:

- języki **kompilowane** – kod źródłowy jest tłumaczony na kod maszynowy za pomocą kompilatora (ANSI C, C++, FORTRAN i in.),
- języki **interpretowane** – kod źródłowy jest wykonywany bezpośrednio przez interpreter (Bash, Python, Perl, PHP i in.).

Języki kompilowane i interpretowane

Jeden z możliwych podziałów języków programowania pozwala wyodrębnić:

- języki **kompilowane** – kod źródłowy jest tłumaczony na kod maszynowy za pomocą kompilatora (ANSI C, C++, FORTRAN i in.),
- języki **interpretowane** – kod źródłowy jest wykonywany bezpośrednio przez interpreter (Bash, Python, Perl, PHP i in.).

Narzędzia programistyczne:

- edytor tekstu, kompilator (w tym linker)/interpreter, debugger,
- opcjonalnie: środowisko programistyczne zawierające funkcjonalność wszystkich powyższych narzędzi (i więcej).

Przykładowe narzędzia:

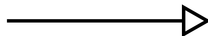
- vim, emacs (edytory),
- gcc, gfortran (kompilatory); gdb (debugger).

Języki kompilowane i interpretowane

KOD ŹRÓDŁOWY

```
#include <stdio.h>
int main (void)
{
  puts ("Hello World!");
  return 0;
}
```

KOMPILACJA

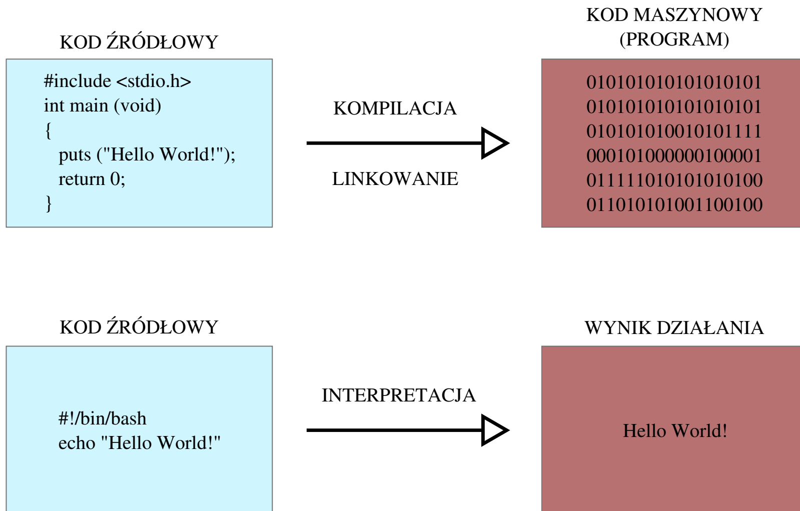


LINKOWANIE

KOD MASZYNOWY
(PROGRAM)

```
0101010101010101010101
0101010101010101010101
0101010100101011111
000101000000100001
011111010101010100
011010101001100100
```

Języki kompilowane i interpretowane



Uruchamianie skryptów powłoki

```
student@wftlab-180:~$
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$ file hello.sh
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$ file hello.sh
hello.sh: Bourne-shell script, ASCII text executable
student@wftlab-180:~$
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$ file hello.sh
hello.sh: Bourne-shell script, ASCII text executable
student@wftlab-180:~$ ./hello.sh
```

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$ file hello.sh
hello.sh: Bourne-shell script, ASCII text executable
student@wftlab-180:~$ ./hello.sh
Hello World!
student@wftlab-180:~$
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$ file hello.sh
hello.sh: Bourne-shell script, ASCII text executable
student@wftlab-180:~$ ./hello.sh
Hello World!
student@wftlab-180:~$ ls -l hello.sh
```

Uruchamianie skryptów powłoki

```
student@wftlab-180:~$ cat hello.sh
#!/bin/bash
echo "Hello World!"
student@wftlab-180:~$ file hello.sh
hello.sh: Bourne-shell script, ASCII text executable
student@wftlab-180:~$ ./hello.sh
Hello World!
student@wftlab-180:~$ ls -l hello.sh
-rwxr-xr-x 1 student student 33 paź 22 12:13 hello.sh
student@wftlab-180:~$
```

Shebang line – linia specjalna:

```
#!/bin/bash
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
student@wftlab-180:~/zadanie1$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
student@wftlab-180:~/zadanie1$ ls -l main.c
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy należą do grup. Prawa dostępu mogą dotyczyć indywidualnych użytkowników, całych grup lub wszystkich pozostałych użytkowników.

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy należą do grup. Prawa dostępu mogą dotyczyć indywidualnych użytkowników, całych grup lub wszystkich pozostałych użytkowników.

Format praw dostępu – 4 bloki

```
- rwx rwx rwx
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~$ cd zadanie1
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy należą do grup. Prawa dostępu mogą dotyczyć indywidualnych użytkowników, całych grup lub wszystkich pozostałych użytkowników.

Format praw dostępu – 4 bloki

`-   rwx   rwx   rwx`

- 1. blok (-): znak specjalny (plik/katalog),
- 2. blok (`rwx`): uprawnienia właściciela pliku,
- 3. blok (`rwx`): uprawnienia użytkowników w grupie właściciela,
- 4. blok (`rwx`): uprawnienia wszystkich pozostałych użytkowników.

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r main.c
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r main.c
student@wftlab-180:~/zadanie1$
```

Brak komunikatu oznacza: OK – zrobione!

```
student@wftlab-180:~/zadanie1$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r main.c
student@wftlab-180:~/zadanie1$
```

Brak komunikatu oznacza: OK – zrobione!

```
student@wftlab-180:~/zadanie1$ ls -l main.c
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r main.c
student@wftlab-180:~/zadanie1$
```

Brak komunikatu oznacza: OK – zrobione!

```
student@wftlab-180:~/zadanie1$ ls -l main.c
--w------- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Użytkownicy, konta i grupy

```
student@wftlab-180:~/zadanie1$ ls -l main.c
-rw-r--r-- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$ chmod -r main.c
student@wftlab-180:~/zadanie1$
```

Brak komunikatu oznacza: OK – zrobione!

```
student@wftlab-180:~/zadanie1$ ls -l main.c
--w------- 1 student student 0 wrz 1 23:20 main.c
student@wftlab-180:~/zadanie1$
```

Uruchamianie skryptów:

tworząc nowy skrypt będziemy najczęściej nadawać mu prawo wykonywalności, choć nie jest to konieczne (o tym później).

Użytkownicy, konta i grupy

Grupy uprawnień:

- `u` (*owner*)
- `g` (*group*)
- `o` (*others*)
- `a` (*all users*)

Użytkownicy, konta i grupy

Grupy uprawnień:

- `u` (*owner*)
- `g` (*group*)
- `o` (*others*)
- `a` (*all users*)

Typy uprawnień:

- `r` (4)
- `w` (2)
- `x` (1)

Użytkownicy, konta i grupy

Grupy uprawnień:

- `u` (*owner*)
- `g` (*group*)
- `o` (*others*)
- `a` (*all users*)

Typy uprawnień:

- `r` (4)
- `w` (2)
- `x` (1)

Przykłady:

- `chmod 640 plik.txt`
- `chmod 740 plik.txt`
- `chown nazwa.grupa plik.txt`

Użytkownicy, konta i grupy

Grupy uprawnień:

- u (*owner*)
- g (*group*)
- o (*others*)
- a (*all users*)

Typy uprawnień:

- r (4)
- w (2)
- x (1)

Przykłady:

- `chmod 640 plik.txt`
- `chmod 740 plik.txt`
- `chown nazwa.grupa plik.txt`

Uwagi:

Jakie powinny być prawa dostępu do katalogów domowych użytkowników, a jakie do plików konfiguracyjnych systemu?

Pętla `for`:

Pętla `for`:

```
for i in 1 2 3 4 5
do
  echo $i
done
```

Pętla `for`:

Pętla `for`:

```
for i in 1 2 3 4 5
do
  echo $i
done
```

Wynik działania:

1
2
3
4
5

Pętla `for`:

Pętla `for`:

```
for i in 1 2 3 4 5
do
  echo $i
done
```

Wynik działania:

1
2
3
4
5

Dla każdego elementu `i` z zadanego zbioru wykonaj instrukcje zawarte pomiędzy `do` a `done`. Zmienne wywołujemy poprzedzając je znakiem $.

Pętla `for`:

Można inaczej:

```
$ seq 1 1 5
```

Pętla for:

Można inaczej:

```
$ seq 1 1 5
1
2
3
4
5
```

Pętla for:

Można inaczej:

```
$ seq 1 1 5
1
2
3
4
5
```

SKŁADNIA: `seq [początek] [krok] [koniec]`

Pętla for:

Można inaczej:

```
$ seq 1 1 5
1
2
3
4
5
```

SKŁADNIA: `seq [początek] [krok] [koniec]`

A w skrypcie (symbol ‘ pod klawiszem ESC):

```
for i in ‘seq 1 1 5‘
do
  echo $i
done
```

Pętla `for`:

Nie muszą to być liczby całkowite:

```
$ seq 0.0 0.5 2
0.0
0.5
1.0
1.5
2.0
```

Pętla `for`:

Nie muszą to być liczby całkowite:

```
$ seq 0.0 0.5 2
0.0
0.5
1.0
1.5
2.0
```

Uwaga na separator dziesiętny – zależy od lokalizacji środowiska (przecinek/kropka). Można np.: `export LC_NUMERIC=C`.

Pętla for:

Nie muszą to być liczby całkowite:

```
$ seq 0.0 0.5 2
0.0
0.5
1.0
1.5
2.0
```

Uwaga na separator dziesiętny – zależy od lokalizacji środowiska (przecinek/kropka). Można np.: `export LC_NUMERIC=C`.

```
for ((i=0; i<10; i++))
do
  echo $i
done
```

Pętla `while`:

Pętla `while`:

```
x=1
while [ $x -le 4 ]
do
  echo "x=$x"
  x=$[x + 1]
done
```

Pętla while:

Pętla while:

```
x=1
while [ $x -le 4 ]
do
  echo "x=$x"
  x=$[x + 1]
done
```

Wynik działania:

x=1
x=2
x=3
x=4

Pętla while:

Pętla while:

```
x=1
while [ $x -le 4 ]
do
  echo "x=$x"
  x=$[x + 1]
done
```

Wynik działania:

x=1
x=2
x=3
x=4

Wykonuj instrukcje w pętli dopóki `[ warunek ]` pozostaje spełniony.

Pętla `until`:

Pętla `until`:

```
x=1
until [ $x -gt 4 ]
do
  echo "x=$x"
  x=$[x + 1]
done
```

Pętla until:

Pętla until:

```
x=1
until [ $x -gt 4 ]
do
  echo "x=$x"
  x=$[x + 1]
done
```

Wynik działania:

x=1
x=2
x=3
x=4

Pętla `until`:

Pętla `until`:

```
x=1
until [ $x -gt 4 ]
do
  echo "x=$x"
  x=$[x + 1]
done
```

Wynik działania:

x=1
x=2
x=3
x=4

Wykonuj instrukcje w pętli dopóki `[ warunek ]` pozostaje **nie**spełniony.

Instrukcja warunkowa `if`

Konstrukcje typu [warunek] (spacje przed/po są konieczne!) realizują instrukcję `test` (`man test`).

Na przykład:

```
if [ -e plik ]
then
  operacja1
else
  operacja2
fi
```

Jeżeli *plik* istnieje – wykonaj operację 1, jeżeli nie istnieje – wykonaj operację 2. Można zadać więcej warunków poprzez `elif [ warunek ]`. Instrukcje `elif` i `else` są opcjonalne. Wszystkie możliwe warunki można sprawdzić w dokumentacji polecenia `test` (`man test`).

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$ c=`expr 5 + 5`
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$ c=`expr 5 + 5`
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$ c=`expr 5 + 5`
student@wftlab-180:~$ echo $c
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$ c=`expr 5 + 5`
student@wftlab-180:~$ echo $c
10
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$ c=‘expr 5 + 5‘
student@wftlab-180:~$ echo $c
10
student@wftlab-180:~$ let c=‘expr $c + 5‘
```

Arytmetyka w Bashu

```
student@wftlab-180:∼$ a=1
student@wftlab-180:∼$ echo $a
1
student@wftlab-180:∼$ echo $((2+2))
4
student@wftlab-180:∼$ b=2
student@wftlab-180:∼$ let b=$b+2
student@wftlab-180:∼$ echo $b
5
student@wftlab-180:∼$ expr 5 + 5
10
student@wftlab-180:∼$ c=‘expr 5 + 5‘
student@wftlab-180:∼$ echo $c
10
student@wftlab-180:∼$ let c=‘expr $c + 5‘; echo $c
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ a=1
student@wftlab-180:~$ echo $a
1
student@wftlab-180:~$ echo $((2+2))
4
student@wftlab-180:~$ b=2
student@wftlab-180:~$ let b=$b+2
student@wftlab-180:~$ echo $b
5
student@wftlab-180:~$ expr 5 + 5
10
student@wftlab-180:~$ c=‘expr 5 + 5‘
student@wftlab-180:~$ echo $c
10
student@wftlab-180:~$ let c=‘expr $c + 5‘; echo $c ; 15
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++))
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++)); echo $d
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++)); echo $d
31
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++)); echo $d
31
student@wftlab-180:~$
```

Możliwa jest prosta arytmetyka w powłoce Bash, ale możemy też użyć zewnętrznego narzędzia do wykonywania operacji matematycznych, np. `bc` (więcej informacji: `man bc`).

```
student@wftlab-180:~$ echo 13/2 | bc
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++)); echo $d
31
student@wftlab-180:~$
```

Możliwa jest prosta arytmetyka w powłoce Bash, ale możemy też użyć zewnętrznego narzędzia do wykonywania operacji matematycznych, np. `bc` (więcej informacji: `man bc`).

```
student@wftlab-180:~$ echo 13/2 | bc
6
student@wftlab-180:~$
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++)); echo $d
31
student@wftlab-180:~$
```

Możliwa jest prosta arytmetyka w powłoce Bash, ale możemy też użyć zewnętrznego narzędzia do wykonywania operacji matematycznych, np. `bc` (więcej informacji: `man bc`).

```
student@wftlab-180:~$ echo 13/2 | bc
6
student@wftlab-180:~$ echo 13/2 | bc -l
```

Arytmetyka w Bashu

```
student@wftlab-180:~$ d=20
student@wftlab-180:~$ ((d = $d + 10))
student@wftlab-180:~$ echo $d
30
student@wftlab-180:~$ ((d++)); echo $d
31
student@wftlab-180:~$
```

Możliwa jest prosta arytmetyka w powłoce Bash, ale możemy też użyć zewnętrznego narzędzia do wykonywania operacji matematycznych, np. `bc` (więcej informacji: `man bc`).

```
student@wftlab-180:~$ echo 13/2 | bc
6
student@wftlab-180:~$ echo 13/2 | bc -l
6.50000000000000000000
student@wftlab-180:~$
```

Architektury wieloprocesorowe

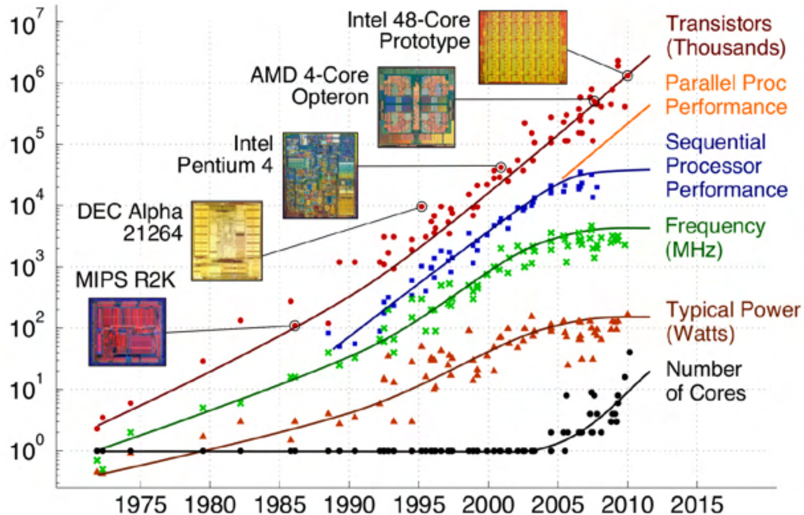
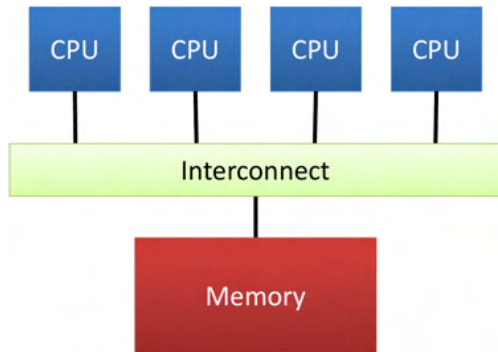
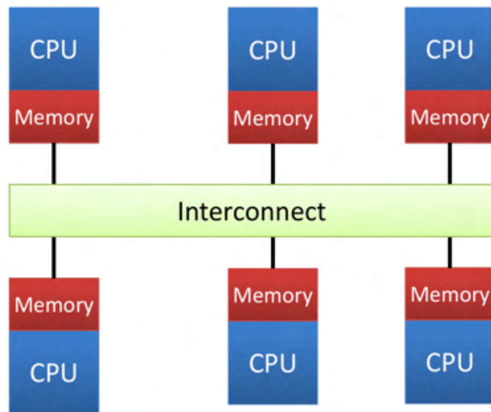


Figure: Christopher Batten, ECE 5950 Complex Digital ASIC Design Course Overview

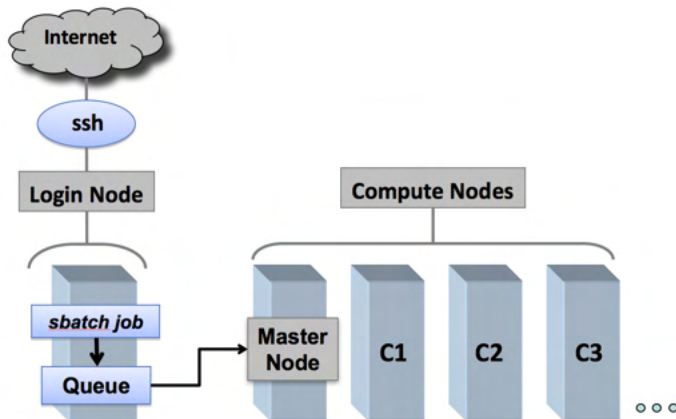
Pamięć współdzielona i rozproszona





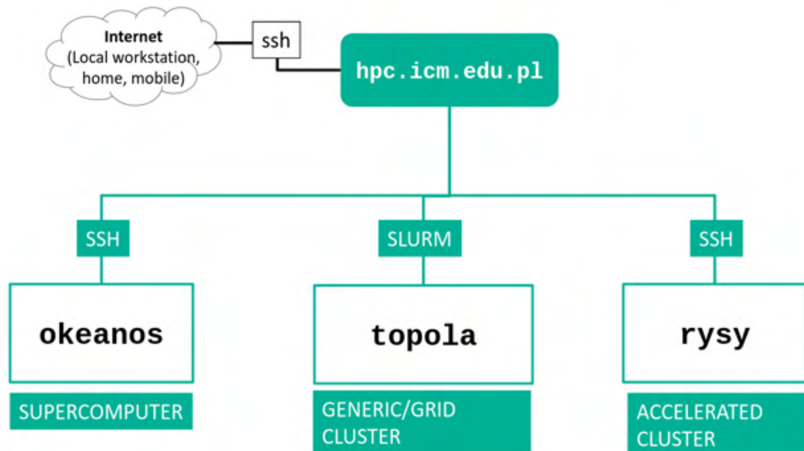
Źródło: **R. Busseuil** et al. `DOI: 10.1007/978-3-642-28566-0_10`

Infrastruktura HPC



Cornell Virtual Workshop (https://cvw.cac.cornell.edu/environment/slurm_intro)

Infrastruktura ICM



System kolejkowy:

- Alokuje zasoby (węzły, CPU, pamięć operacyjna) dla zadań obliczeniowych,
- uruchamia zadania równoległe na zaalokowanych zasobach,
- dba o równomierne obciążenie maszyn i równy przydział zasobów.

Slurm: System kolejkowy

System kolejkowy:

- Alokuje zasoby (węzły, CPU, pamięć operacyjna) dla zadań obliczeniowych,
- uruchamia zadania równoległe na zaalokowanych zasobach,
- dba o równomierne obciążenie maszyn i równy przydział zasobów.

Pojęcia:

- węzeł (ang. *node*) – pojedyncza jednostka obliczeniowa wyposażona w CPU i pamięć),
- partycja (and. *partition* – zbiór węzłów o określonej charakterystyce,
- zadanie (ang. *job*) – pojedyncze zadanie obliczeniowe.

Slurm: System kolejkowy

System kolejkowy:

- Alokuje zasoby (węzły, CPU, pamięć operacyjna) dla zadań obliczeniowych,
- uruchamia zadania równoległe na zaalokowanych zasobach,
- dba o równomierne obciążenie maszyn i równy przydział zasobów.

Pojęcia:

- węzeł (ang. *node*) – pojedyncza jednostka obliczeniowa wyposażona w CPU i pamięć),
- partycja (and. *partition* – zbiór węzłów o określonej charakterystyce,
- zadanie (ang. *job*) – pojedyncze zadanie obliczeniowe.

Przykłady nazw partycji: *topola*, *okeanos* (domyślna), *gpu*, *ve*.

Slurm: System kolejkowy

Wybrane instrukcje:

- `sbatch` – umieszcza zadanie (skrypt zadania) w kolejce (alokuje zasoby),
- `srun` – uruchamia zadanie równoległe w ramach alokacji,
- `scancel` – usuwa zadanie z kolejki,
- `squeue` – wyświetla stan kolejki,
- `sinfo` – wyświetla informacje o węzłach i partycjach,
- `scontrol` – wyświetla szczegóły (lub modyfikuje) zadanie.

Slurm: System kolejkowy

Sesja interaktywna:

```
srun -A nr-alokacji -p topola --nodes=1 --ntasks-per-node=1 --pty bash -l
```

Slurm: System kolejkowy

Sesja interaktywna:

```
srun -A nr-alokacji -p topola --nodes=1 --ntasks-per-node=1 --pty bash -l
```

Tryb wsadowy – przykładowy skrypt:

```
#!/bin/bash -l
#SBATCH -J nazwa
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --mem 1000
#SBATCH --time=1:00:00
#SBATCH -A <Grant_ID>
#SBATCH -p topola
./program
```

Slurm: System kolejkowy

Sesja interaktywna:

```
srun -A nr-alokacji -p topola --nodes=1 --ntasks-per-node=1 --pty bash -l
```

Tryb wsadowy – przykładowy skrypt:

```
#!/bin/bash -l
#SBATCH -J nazwa
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --mem 1000
#SBATCH --time=1:00:00
#SBATCH -A <Grant_ID>
#SBATCH -p topola
./program
```

Zlecenie zadania:

```
sbatch job.sl
```

Anulowanie zadania:

```
scancel 1234567
```

Szczegóły zadania:

```
scontrol show job 1234567
```

Slurm: System kolejkowy

Listing zadań w kolejce (squeue):

```
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
5328766 topola hfor_LaB mariana PD 0:00 10 (Resources)
5328767 topola hfor_LaB mariana PD 0:00 10 (Priority)
5328768 topola hfor_LaB mariana PD 0:00 10 (Priority)
5328769 topola hfor_LaB mariana PD 0:00 10 (Priority)
5329499 topola snpk8_1 purpurea R 9:46:09 1 t7-15
5329498 topola snpk7_5 purpurea R 9:46:32 1 t7-15
5329497 topola snpk7_4 purpurea R 9:46:39 1 t7-15
5329496 topola snpk7_3 purpurea R 9:46:45 1 t7-15
5328765 topola hfor_LaB mariana R 30:01 10 t9-[7-8,12],t10-[2-3]
```