Kurs administrowania systemem Linux

Zajęcia nr 4: Powłoka systemowa (dokończenie). Rootfs

Instytut Informatyki Uniwersytetu Wrocławskiego

22 marca 2022

Deskryptory plików w Linuksie

- Deskryptory: małe liczby nieujemne identyfikujące otwarte pliki procesu.
- Deskryptory 0, 1 i 2 to standardowe strumienie wejściowy, wyjściowy i błędów.
- Linki symboliczne w katalogach `/proc/PID/fd`.
- Linki symboliczne w katalogu `/proc/self/fd`.
- Uwaga: `/proc/self` jest linkiem sybolicznym do `/proc/$PID`. Każdy proces „widzi” ten link inaczej.
- `procfs` jest symulowanym przez jądro systemem plików zamontowanym w katalogu `/proc`.
- Dodatkowe statyczne linki symboliczne z psudosystemu plików `udev` zamontowanego w katalogu `/dev`:

```
/dev/fd -> /proc/self/fd
/dev/stdin -> /proc/self/fd/0
/dev/stdout -> /proc/self/fd/1
/dev/stderr -> /proc/self/fd/2
```

Przekierowania

```
$ /bin/ls -l /proc/self/fd/
```

```
total 0
lrwx------ 1 64 Mar 10 19:18 0 -> /dev/pts/4
lrwx------ 1 64 Mar 10 19:18 1 -> /dev/pts/4
lrwx------ 1 64 Mar 10 19:18 2 -> /dev/pts/4
lr-x------ 1 64 Mar 10 19:18 3 -> /proc/93546/fd
```

Przekierowania

```
$ /bin/ls -l /proc/self/fd/ < /home/user/myfile.txt
```

```
total 0
lrwx------ 1 64 Mar 10 19:18 0 -> /home/user/myfile.txt
lrwx------ 1 64 Mar 10 19:18 1 -> /dev/pts/4
lrwx------ 1 64 Mar 10 19:18 2 -> /dev/pts/4
lr-x------ 1 64 Mar 10 19:18 3 -> /proc/93546/fd
```

Przekierowania

```
$ /bin/ls -l /proc/self/fd/ | cat
```

```
total 0
lrwx------ 1 64 Mar 10 19:18 0 -> /dev/pts/4
lrwx------ 1 64 Mar 10 19:18 1 -> pipe:[36547]
lrwx------ 1 64 Mar 10 19:18 2 -> /dev/pts/4
lr-x------ 1 64 Mar 10 19:18 3 -> /proc/93546/fd
```

Składnia przekierowań

Otwarcie pliku (urządzenia) do zapisu i/lub odczytu

`d>` *plik-lub-urządzenie*
`d<` *plik-lub-urządzenie*
`d<>` *plik-lub-urządzenie*

Otwarcie pliku do odczytu w trybie dołączania

`d>>` *plik*

Skopiowanie istniejącego deskryptora

`d>& d'` oraz `d<& d'`

Przeniesienie istniejącego deskryptora

`d>& d'-` oraz `d<& d'-`

Zamknięcie deskryptora

`d>& -` oraz `d<& -`

Domyślnie $d = 0$ dla < oraz $d = 1$ dla > i >>.

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — ?
2. `echo Message 2>/dev/null >&2` —
3. `echo Message 2>&1 2>/dev/null` —
4. `echo Message 2>&1- 2>/dev/null` —

```
$ echo Message >&2 2>/dev/null
```

???

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` —
3. `echo Message 2>&1 2>/dev/null` —
4. `echo Message 2>&1- 2>/dev/null` —

```
$ echo Message >&2 2>/dev/null
```

```
Message
```

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` — ?
3. `echo Message 2>&1 2>/dev/null` —
4. `echo Message 2>&1- 2>/dev/null` —

```
$ echo Message 2>/dev/null >&2
```

???

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` — nie ma
3. `echo Message 2>&1 2>/dev/null` —
4. `echo Message 2>&1- 2>/dev/null` —

```
$ echo Message 2>/dev/null >&2
```

nic nie zostanie wypisane na konsoli

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` — nie ma
3. `echo Message 2>&1 2>/dev/null` — ?
4. `echo Message 2>&1- 2>/dev/null` —

```
$ echo Message 2>&1 2>/dev/null
```

???

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` — nie ma
3. `echo Message 2>&1 2>/dev/null` — jest
4. `echo Message 2>&1- 2>/dev/null` —

```
$ echo Message 2>&1 2>/dev/null
```

```
Message
```

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` — nie ma
3. `echo Message 2>&1 2>/dev/null` — jest
4. `echo Message 2>&1- 2>/dev/null` — ?

```
$ echo Message 2>&1- 2>/dev/null
```

???

Semantyka przekierowań

1. `echo Message >&2 2>/dev/null` — jest
2. `echo Message 2>/dev/null >&2` — nie ma
3. `echo Message 2>&1 2>/dev/null` — jest
4. `echo Message 2>&1- 2>/dev/null` — nie ma

```
$ echo Message 2>&1- 2>/dev/null
```

nic nie zostanie wypisane na konsoli

Dostęp procesu do plików w Linuksie

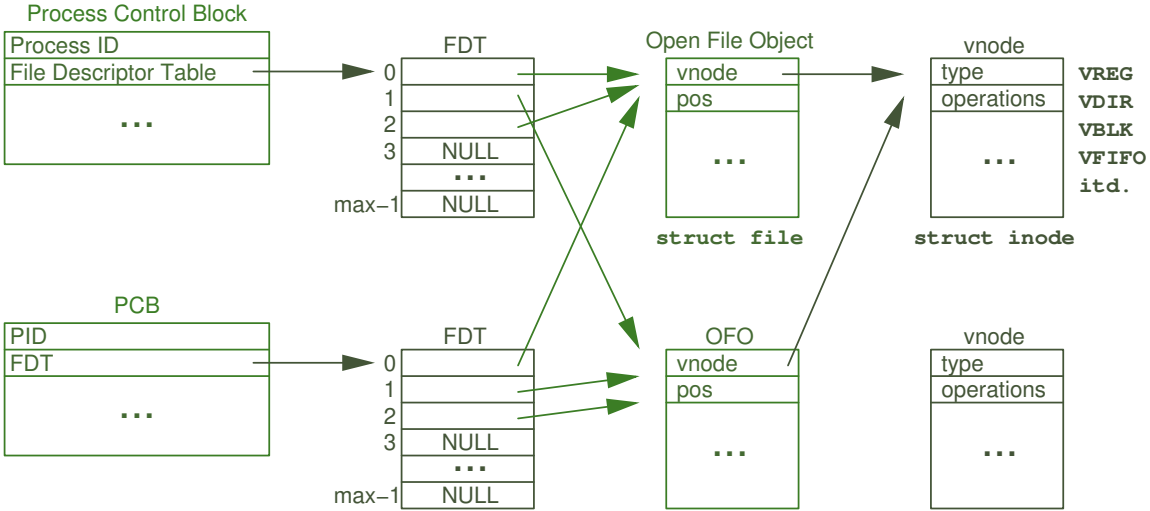
Metadane każdego procesu są przechowywane w strukturze *Process Control Block* (PCB) w *tablicy procesów*, zawierającej m. in.:

- ID procesu (PID),
- stan procesu,
- tablicę otwartych plików: *File Descriptor Table* (FDT).

Informacje o otwartych plikach:

- FDT to tablica wskaźników na struktury *Open File Object* (w Linuksie `struct file`) przechowywane w tablicy otwartych plików.
- FDT jest indeksowana liczbami naturalnymi 0, 1, 2,... zwanymi *deskryptorami plików* danego procesu.
- OFO zawiera wskaźnik na v-node (virtual node), tryb dostępu, pozycję w pliku itp.
- v-node opisuje jeden konkretny plik w systemie plików.
- v-node'y istnieją tylko dla plików będących w użyciu.
- Wiele OFO może wskazywać na ten sam v-node (na ten sam plik).

Deskryptory plików w Linuksie



Operacje na plikach

`int open(const char *pathname, int flags)`

- Wyszukuje v-node pliku `pathname`. Jeśli go nie ma, to tworzy go w pamięci.
- Tworzy nowy OFO wskazujący na ten v-node i ustala jego tryb na `flags` (`O_RDONLY`, `O_WRONLY`, `O_RDWR` itp.).
- Zwiększa refcount v-node'a.
- Znajduje wolny deskryptor w FDT i wstawia tam wskaźnik na utworzony OFO.
- Zwraca deskryptor pliku.

`int close(int fd)`

- Zmniejsza refcount OFO wskazywanego przez `fd`.
- Jeśli spadł do zera, to usuwa OFO i zmniejsza refcount v-noda wskazywanego przez OFO.
- Jeśli spadł do zera, to usuwa v-node'a.
- Wpisuje `NULL` w pozycji `fd` w tablicy FDT procesu.

Kopiowanie deskryptorów i refcount OFO

Dlaczego refcount OFO może być większy niż 1?

- `fork` tworzy kopię PCB i kopię FDT.
- `dup` kopiuje deskryptory plików.

```
int dup(int oldfd)
int dup2(int oldfd, int newfd)
int dup3(int oldfd, int newfd, int flags)
```

- `dup` znajduje wolny deskryptor w FD i kopiuje wpis `oldfd` w FDT.
- `dup2` kopiuje element `oldfd` tablicy FDT do wskazanego elementu `newfd`.
- `dup3` modyfikuje dodatkowo flagę *close-on-exec* elementu `newfd` tablicy FDT.

Jak bash przygotowuje podproces do wykonania?

Utwórz `fork`-iem proces potomny i przekaż mu do wykonania instrukcję z przekierowaniami.

Wykonanie instrukcji z przekierowaniami w podprocesie:

- Zamknij wszystkie deskryptory powyżej 2.
- Zmodyfikuj deskryptory 0–2 zgodnie z charakterem instrukcji.
- Wykonaj wszystkie przekierowania od lewej do prawej.
- Wykonaj `exec`.

Implementacja przekierowań

`n > file`

```
int k = open(file, O_WRONLY);
dup2(k,n);
close(k);
```

Implementacja przekierowań

`n < file`

```
int k = open(file, O_RDONLY);
dup2(k,n);
close(k);
```

Implementacja przekierowań

`n <> file`

```
int k = open(file, O_RDWR);
dup2(k,n);
close(k);
```

Implementacja przekierowań

`n >> file`

```
int k = open(file, O_APPEND);
dup2(k,n);
close(k);
```

Implementacja przekierowań

`n >> file`

```
int k = open(file, O_APPEND);
dup2(k,n);
close(k);
```

`n >& m`

```
dup2(m,n)
```

Implementacja przekierowań

`n >> file`

```
int k = open(file, O_APPEND);
dup2(k,n);
close(k);
```

`n >&- m`

```
dup2(m,n)
close(m)
```

Implementacja przekierowań

```
n >> file
```

```
int k = open(file, O_APPEND);
dup2(k,n);
close(k);
```

```
n >& -
```

```
close(n)
```

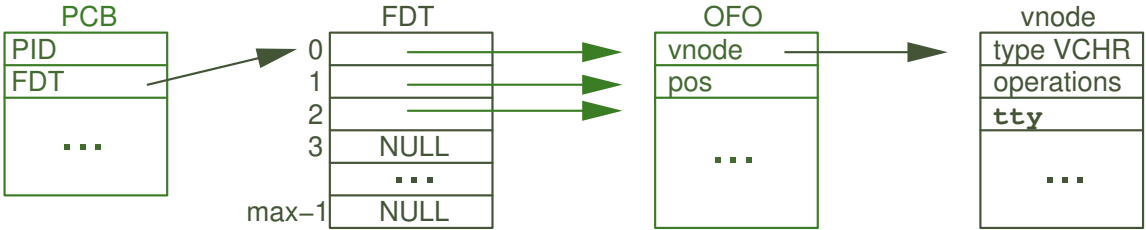
Przykład z wczorajszej pracowni

```
RESULT=$(program arg1 ... argn 3>&1 1>&2 2>&3)
```

Wykonanie tej konstrukcji:

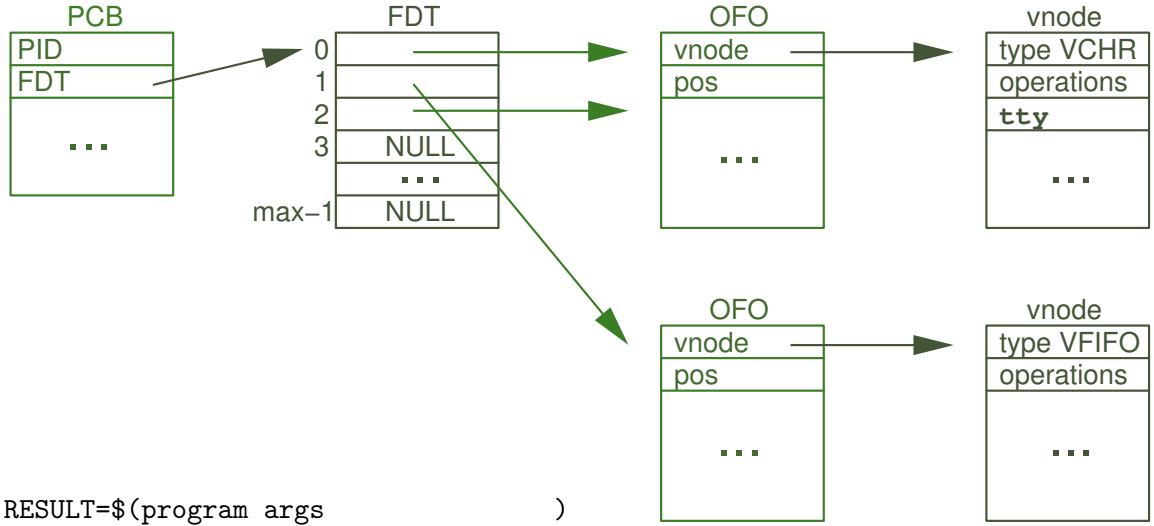
- bash tworzy nowy potok (*fifo*),
- standardowe wyjście `program`-u jest przekierowane do tego potoku,
- bash czyta zawartość tego potoku i zapisuje do zmiennej środowiskowej `RESULT`.

Zamiana deskryptorów miejscami



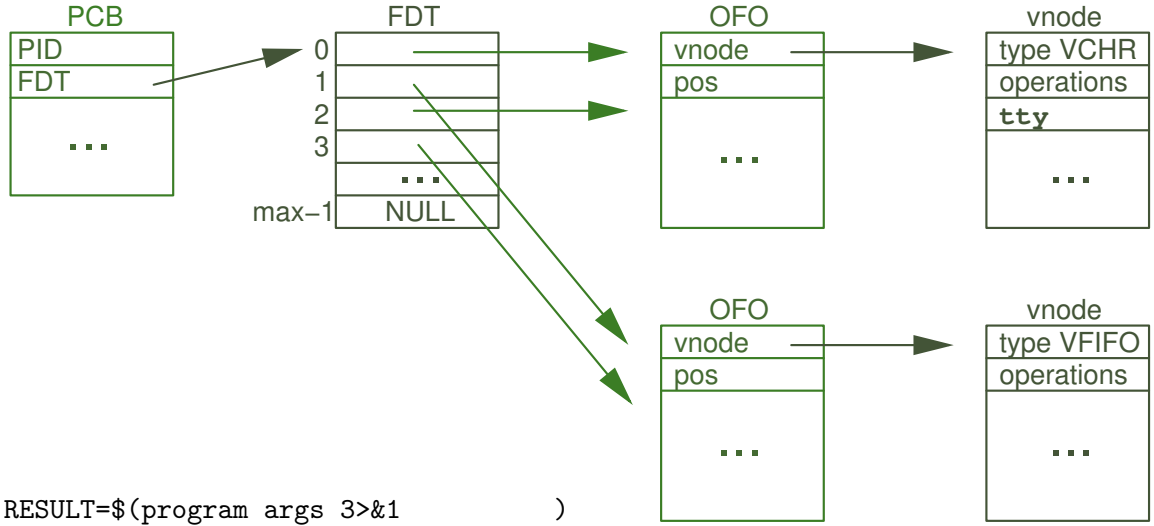

```
program args
```

Zamiana deskryptorów miejscami



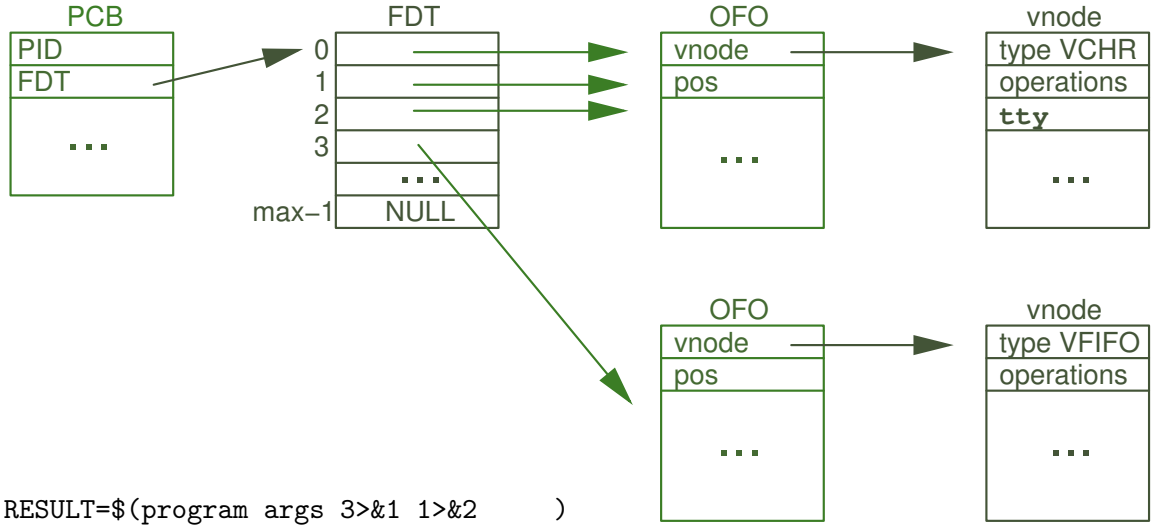

```
RESULT=$(program args                    )
```

Zamiana deskryptorów miejscami



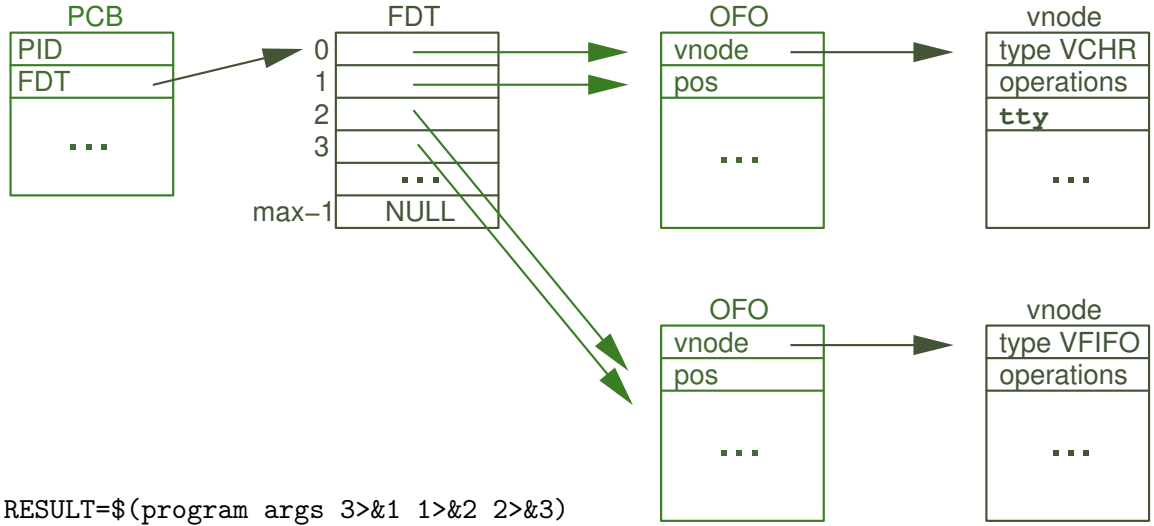

```
RESULT=$(program args 3>&1                )
```

Zamiana deskryptorów miejscami




```
RESULT=$(program args 3>&1 1>&2      )
```

Zamiana deskryptorów miejscami




```
RESULT=$(program args 3>&1 1>&2 2>&3)
```

Eksperymenty (1)

```
$ echo $(ls -lAF /proc/self/fd/)
total 0 lrwx------ 1 user user 64 Mar 22 17:15 0 -> /dev/pts/1 l-wx------ 1
user user 64 Mar 22 17:15 1 -> pipe:[33264] lrwx------ 1 user user 64 Mar
22 17:15 2 -> /dev/pts/1 lr-x------ 1 user user 64 Mar 22 17:15 3 ->
/proc/7321/fd/
```

Czytelniej:

```
total 0
lrwx------ 1 user user 64 Mar 22 17:15 0 -> /dev/pts/1
l-wx------ 1 user user 64 Mar 22 17:15 1 -> pipe:[33264]
lrwx------ 1 user user 64 Mar 22 17:15 2 -> /dev/pts/1
lr-x------ 1 user user 64 Mar 22 17:15 3 -> /proc/7321/fd/
```

Eksperymenty (2)

```
$ echo $(ls -lAF /proc/$$/fd/)
total 0 lrwx------ 1 user user 64 Mar 22 10:49 0 -> /dev/pts/1 lrwx------ 1
user user 64 Mar 22 17:19 1 -> /dev/pts/1 lrwx------ 1 user user64 Mar 22
17:19 2 -> /dev/pts/1 lrwx------ 1 user user 64 Mar 22 17:19 255 ->
/dev/pts/1 lr-x------ 1 user user 64 Mar 22 17:19 3 -> pipe:[33353]
```

Czytelniej:

```
total 0
lrwx------ 1 user user 64 Mar 22 10:49 0 -> /dev/pts/1
lrwx------ 1 user user 64 Mar 22 17:19 1 -> /dev/pts/1
lrwx------ 1 user user 64 Mar 22 17:19 2 -> /dev/pts/1
lrwx------ 1 user user 64 Mar 22 17:19 255 -> /dev/pts/1
lr-x------ 1 user user 64 Mar 22 17:19 3 -> pipe:[33353]
```

Eksperymenty (3)

```
$ echo $(ls -lAF /proc/self/fd/ 3>&1 1>&2 2>&3)
total 0
lrwx------ 1 user user 64 Mar 22 17:23 0 -> /dev/pts/1
lrwx------ 1 user user 64 Mar 22 17:23 1 -> /dev/pts/1
l-wx------ 1 user user 64 Mar 22 17:23 2 -> pipe:[33390]
l-wx------ 1 user user 64 Mar 22 17:23 3 -> pipe:[33390]
lr-x------ 1 user user 64 Mar 22 17:23 4 -> /proc/7464/fd/
```

Oraz:

```
$ X=$(ls -lAF /AAA 3>&1 1>&2 2>&3)
$ echo $X
/bin/ls: cannot access /AAA: No such file or directory
```

Szczególne przypadki przekierowań

- `exec` *`przekierowania`* pozwala przekierować deskryptory bieżącej powłoki. Np.:

  ```
  exec 2> errors.log
  ```

  powoduje zapisywanie zawartości standardowego strumienia dla błędów we wszystkich dalszych poleceniach do pliku `errors.log`.
- `exec <&-` jest równoważne `exit` (bo powoduje zamknięcie standardowego strumienia wejściowego).
- Jednoczesne przekierowania `stdout` i `stderr`: `&>` i `&>>` (por. `&|`).
  (Także `>&` i `>>&` jeśli następne słowo nie rozwija się do liczby bądź `-`.)
- *Here documents:* `<<` i `<<-`.
- *Here strings:* `<<<`.
- Automatyczne tworzenie nowych deskryptorów: {*zmienna*}[>|<|>>| ...].
- `$(< file)` — szybsza wersja `$(cat file)`.

Przykład: zapisywanie do ustalonego pliku

Otwieranie w trybie do dołączania

```
MYFILE=myfile.txt
echo -n "My "   >> "$MYFILE"
echo -n "long " >> "$MYFILE"
echo "message"  >> "$MYFILE"
```

- Plik jest otwierany i zamykany osobno dla każdego polecenia `echo`.
- Dobre w przypadku *logów*, bo współdziała z `logrotate`.
- Nieefektywne w przypadku masowego zapisywania.

Otwieranie pliku dla wielu poleceń

Ręczny wybór deskryptora

```
exec 3> myfile.txt
echo -n "My "   >&3
echo -n "long " >&3
echo "message"  >&3
exec 3>&-
```

Należy wybierać deskryptory z przedziału 3–9.

Automatyczny wybór deskryptora

```
exec {MYFD}> myfile.txt
echo -n "My "   >& $MYFD
echo -n "long " >& $MYFD
echo "message"  >& $MYFD
exec {MYFD}>&-
```

source

- `source` *`script`* — wykonanie skryptu *`script`* w *bieżącej* powłoce.
- Jedyny sposób, żeby zmienić srodowisko bieżącego procesu.
- Skrypty startowe `/etc/bash.bashrc`, `/etc/profile`, `~/.bashrc` itp. są `source`'owane.
- Biblioteki funkcji wykorzystywanych przez wiele skryptów,
  zob. np. `/lib/lsb/init-functions`.
- Skrót: `.` (kropka).
- Uwaga na zaśmiecanie środowiska! Zmienne lokalne dla skryptu powinny zostać `unset`.
  Uwaga na kolizje zmiennych lokalnych skryptu ze zmiennymi globalnymi.

Pliki source'owane a wykonywane (w podprocesie)

Plik source'owany (np. .profile)

```
...
TMP="zmienna lokalna"
ORIG_IFS=$IFS
IFS=$' \t'
...
IFS=$ORIG_IFS
unset ORIG_IFS TMP
```

Uwaga na zmienne istniejące już w środowisku (np. TMP)!

Plik wykonywany w podprocesie (skrypt)

Modyfikacje zmiennych środowiskowych są ograniczone tylko do wykonania tego skryptu.

Pliki konfiguracyjne skryptów

Skrypt

```
...
MYCONFIG=~/.myconfig
VAR1="script default for VAR1"
VAR2="script default for VAR2"
[ -f $MYCONFIG ] && source $MYCONFIG
...
```

Plik ~/.myconfig

```
# Opis zmiennej VAR1 i zakomentowana wartość domyślna
# VAR1="script default for VAR1"
# Opis zmiennej VAR2 i zakomentowana wartość domyślna
# VAR2="script default for VAR2"
```

Ten sam trik można stosować dla programów:

```
[ -x /usr/bin/mandb ] && /usr/bin/mandb
```

Sortowanie numeryczne nazw

Numerowanie plików

```
N=1
for FILE in *.jpg; do
   mv $FILE pic$N.jpg
   ((N++))
done
```

- W wielu przypadkach nazwy plików są domyślnie sortowane alfabetycznie.
- Nazwa `pic10.jpg` leksykograficznie poprzedza `pic1.jpg`.

Sortowanie numeryczne nazw

Numerowanie plików

```
N=1
for FILE in *.jpg; do
   mv $FILE pic$(printf '%03d' $N).jpg
   ((N++))
done
```

- W wielu przypadkach nazwy plików są domyślnie sortowane alfabetycznie.
- Nazwa `pic10.jpg` leksykograficznie poprzedza `pic1.jpg`.
- Porządek leksykograficzny i numeryczny na literałach o stałej długości z zerami wiodącymi są zgodne.
- Opcja `-v` programu `ls` włącza numeryczne sortowanie nazw z numerami wersji.
- Wzorzec `{000..200}` rozwija się do `000 001 002...`.

Opcje i argumenty programów

Podejście tradycyjne

- Opcje jednoliterowe (np. `-l`), ew. z argumentem.
- Opcja `--` lub `-` — jawne oznaczenie końca opcji.
- Opcje powinny poprzedzać argumenty programu.
- Program `getopt(1)` pozwala na mieszanie opcji i argumentów oraz „zwijanie" opcji (np. `shred -vzun0 file`).

Podejście współczesne

- Opcje mogą być krótkie (jednoliterowe, z pojedynczym `-`) i długie (z podwójnym `--`) lub tylko długie.
- Jeśli tylko długie, to mogą się zaczynać od `-` lub `--`.
- Składnia długich opcji z argumentem: `--opcja=argument` lub `--opcja argument`.
- Jeśli opcja może mieć argument opcjonalny, to dopuszczalna jest tylko pierwsza wersja.

Dokumentacja programów

Skąd można się dowiedzieć o opcjach programów?

- `man(1)`, `apropos(1)` i `whatis(1)`
- GNU Info
- Opcja `-h` lub `--help` *niektórych* programów (nie polecamy).

Nie uruchamiaj programów, których opisu nie znasz!

- Opcja `-h` *zwykle* oznacza pomoc.
- Opcja `-v` — *zwykle* oznacza albo *version*, albo *verbose*.
- Jeśli nie można skorzystać z dokumentacji, lepiej próbować długich opcji `--help` i `--version`.

Pułapki

- Opcja `-h` w programach `ls`, `df` itp. oznacza *human readable*.
- `pkill -v xeyes` — opcja `-v` nie oznacza *verbose*, tylko to samo, co w programie `grep`!

Rekord zapisany na karcie Holleritha

Plik jest wirtualizacją dysku

Wiele mechanizmów w komputerach uległo *wirtualizacji* (np. proces jest wynikiem wirtualizacji CPU itp.)

Dawniej (lata 1960-te)

- Rekord — zapis na pojedynczej karcie perforowanej.
- Zbiór rekordów — plik kart.
- Pamięć bębnowa, potem dyskowa — bufor przechowujący plik kart.
- Pamięć dyskową zaczęto nazywać *dyskiem*.
- IBM VSAM (Virtual Storage Access Metod) — możliwość jednoczesnego przechowywania kilku plików (kart) na jednym dysku — wirtualizacja dysku.
- Pojedynczy wirtualny dysk przechowujący plik rekordów (kart) zaczęto nazywać *plikem*.
- Koncepcja *katalogu* (*kartoteki*) — sposób organizacji plików na dysku.

Filozofia Uniksa: wszystko jest plikiem (prawie)

Wszystko jest plikiem

Unix jest *dyskowym systemem operacyjnym*. Przetwarzanie informacji przez jądro i procesy użytkowników polega na otwieraniu (`open(2)`), czytaniu (`read(2)`), pisaniu (`write(2)`) i zamykaniu (`close(2)`) plików. Większość plików jest nazwana, tj. posiada ścieżkę dostępu w drzewie katalogów (*rootfs*).

Rodzaje plików

- zwykłe
- katalogi
- dowiązania symboliczne
- urządzenia znakowe (niebuforowane)
- urządzenia blokowe (buforowane)
- rurociągi (*pipes*, FIFO)
- gniazda lokalne

Filozofia Uniksa: pliki tekstowe

Pliki tekstowe

Tam, gdzie to możliwe, używaj plików tekstowych.

Pliki tekstowe

- dane i wyniki programów
- logi
- pliki konfiguracyjne
- komunikacja między programami

Uniwersalne narzędzia do przetwarzania plików tekstowych

- powłoka (`bash`)
- edytory (`vi`, `emacs`)
- `awk`, `sed`, `tr` i dziesiątki innych narzędzi
- `perl` (*Practical Extraction and Report Language*)
- inne języki skryptowe

Por. zamknięte formaty binarne: pliki można przetwarzać *tylko* za pomocą dedykowanych programów i *tylko* w zakresie określonym przez te programy.

Obsługa systemów plików w Linuksie

- System plików — wirtualizacja urządzenia blokowego.
- Wiele różnych dyskowych systemów plików, np. `ext{2,3,4}`, `reiser{fs,4}`, `ufs`, `jfs`, `xfs`, `hfs`, `minix`, COW: `btrfs`, `zfs`, log-structured: `logfs`, `nilfs2`, `f2fs`, ROM: `iso9660`, `udf`, `cramfs`, `squashfs`, windowsowe: `FAT{12,16,32}`, `exFAT`, `NTFS` i wiele innych.
- Pseudosystemy plików — „symulowane" przez jądro, nie istnieją na fizycznym urządzeniu blokowym: `sysfs`, `procfs`, `udevfs`, `tmpfs`.
- Uwaga: we FreeBSD `procfs` domyślnie nie jest montowany, bo jest uważany za niebezpieczny. Jest montowany np. w trybie kompatybilności z Linuksem.
- Warstwa abstrakcji: `vfs`.
- W działającym systemie uniksowym: wiele systemów plików *zamontowanych* w różnych miejscach *jednego* drzewa katalogów.
- Por. tradycyjnie w Windows osobne urządzenia `A:`, `B:` (napędy dyskietek), `C:`, `D:` itd.