

# Kurs administrowania systemem Linux

## Zajęcia nr 4: Powłoka systemowa: deskryptory plików

Instytut Informatyki Uniwersytetu Wrocławskiego

18 marca 2024

# Deskryptory plików w Linuksie

- Deskryptory: małe liczby nieujemne identyfikujące otwarte pliki procesu.
- Są używane do indeksowania FDT — *tablicy deskryptorów plików*.
- Każdy proces ma własną tablicę deskryptorów dostępną z PCB (*Process Control Block*).
- Element tej tablicy: `FDT[d]` — wskaźnik na strukturę jądra przechowującą informacje o otwartym pliku (`struct file *`) którego deskryptorem jest `d`.
- Dla przestrzeni użytkownika deskryptory są „uchwyty” (identyfikatorami, nazwami) otwartych plików.
- Syscall `getdtablesize(2)` i polecenie Basha `ulimit -n` ujawniają maksymalny rozmiar tablicy deskryptorów procesu (obecnie przeważnie 65536).
- Największy poprawny deskryptor: powyższa liczba minus jeden (przeważnie 65535).  
Najmniejszy: 0.

## Deskryptory plików w Linuksie (2)

- Konwencja: deskryptory 0, 1 i 2 to standardowe strumienie wejściowy, wyjściowy i błędów.
- Te deskryptory konfiguruje procesowi jego rodzic (plus być może dodatkowe).
- Linki symboliczne w katalogach `/proc/PID/fd`.
- Linki symboliczne w katalogu `/proc/self/fd`.
- Uwaga: `/proc/self` jest linkiem symbolicznym 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 pseudosystemu 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
```

## Deskryptory plików w Linuksie (3)

```
$ ls -l /proc/self/fd/  
total 0  
lrwx----- 1 user user 64 Mar 17 12:17 0 -> /dev/pts/1  
lrwx----- 1 user user 64 Mar 17 12:17 1 -> /dev/pts/1  
lrwx----- 1 user user 64 Mar 17 12:17 2 -> /dev/pts/1  
lrwx----- 1 user user 64 Mar 17 12:17 3 -> /proc/3232/fd/
```

## Deskryptory plików w Linuksie (3)

```
$ ls -l /proc/$$/fd/
total 0
lrwx----- 1 user user 64 Mar 16 17:43 0 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 1 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 2 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 255 -> /dev/pts/1
```

## Deskryptory plików w Linuksie (3)

```
$ ulimit -n
65536
$ exec 65535> file1
$ ls -l /proc/$$/fd/
total 0
lrwx----- 1 user user 64 Mar 16 17:43 0 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 1 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 2 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 255 -> /dev/pts/1
l-wx----- 1 user user 64 Mar 17 12:17 65535 -> /home/user/tmp/file1
```

## Deskryptory plików w Linuksie (3)

```
$ ulimit -n
65536
$ exec 65535> file1
$ ls -l /proc/$$/fd/
total 0
lrwx----- 1 user user 64 Mar 16 17:43 0 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 1 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 2 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 17 12:17 255 -> /dev/pts/1
l-wx----- 1 user user 64 Mar 17 12:17 65535 -> /home/user/tmp/file1
```

### Dygresja: symlink(7)

On Linux, the permissions of a symbolic link are not used in any operations; **the permissions are always 0777** (read, write, and execute for all user categories), and can't be changed.

## Deskryptory plików w Linuksie (3)

```
$ exec 65536> file2
```

```
bash: 65536: Bad file descriptor
```

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

```
total 0
```

```
lrwx----- 1 user user 64 Mar 16 17:43 0 -> /dev/pts/1
```

```
lrwx----- 1 user user 64 Mar 17 12:17 1 -> /dev/pts/1
```

```
lrwx----- 1 user user 64 Mar 17 12:17 2 -> /dev/pts/1
```

```
l-wx----- 1 user user 64 Mar 17 12:18 3 -> /home/user/tmp/file2
```

```
lrwx----- 1 user user 64 Mar 17 12:17 255 -> /dev/pts/1
```

```
l-wx----- 1 user user 64 Mar 17 12:17 65535 -> /home/user/tmp/file1
```



## Deskryptory plików w Linuksie (3)

```
$ exec 3>&-  
$ exec 65535>&-  
$ ls -l /proc/$$/fd/  
total 0  
lrwx----- 1 user user 64 Mar 16 17:43 0 -> /dev/pts/1  
lrwx----- 1 user user 64 Mar 17 12:17 1 -> /dev/pts/1  
lrwx----- 1 user user 64 Mar 17 12:17 2 -> /dev/pts/1  
lrwx----- 1 user user 64 Mar 17 12:17 255 -> /dev/pts/1
```

## Dygresja: ojciec i syn w Bashu (proces potomny vs. „podproces”)

```
# echo $$; (echo $$)
```

# Dygresja: ojciec i syn w Bashu (proces potomny vs. „podproces”)

```
# echo $$; (echo $$)
1590
1590
# echo \$$BASHPID = $BASHPID; echo \$\$ = $$; echo \$$PPID = $PPID
$BASHPID = 1079          # ten proces (Bash)
$$ = 1079                # ten proces (Posix)
$PPID = 1057             # ojciec tego procesu (Posix)
# (echo \$$BASHPID = $BASHPID; echo \$\$ = $$; echo \$$PPID = $PPID)
$BASHPID = 1594          # ten podproces (Bash)
$$ = 1079                # rodzic tego podprocesu (Posix)
$PPID = 1057             # ojciec rodzica tego podprocesu (Posix)
# bash -c 'echo \$$BASHPID = $BASHPID; echo \$\$ = $$; echo \$$PPID = $PPID'
$BASHPID = 1595          # ten proces (Bash)
$$ = 1079                # ojciec tego procesu (Posix)
$PPID = 1595             # ten proces (Posix)
```

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

```
total 0
```

```
lrwx----- 1 user user 64 Mar 10 19:18 0 -> /dev/pts/4
```

```
lrwx----- 1 user user 64 Mar 10 19:18 1 -> /dev/pts/4
```

```
lrwx----- 1 user user 64 Mar 10 19:18 2 -> /dev/pts/4
```

```
lr-x----- 1 user user 64 Mar 10 19:18 3 -> /proc/93546/fd
```

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

```
total 0
```

```
lrwx----- 1 user user 64 Mar 10 19:18 0 -> /home/user/myfile.txt
```

```
lrwx----- 1 user user 64 Mar 10 19:18 1 -> /dev/pts/4
```

```
lrwx----- 1 user user 64 Mar 10 19:18 2 -> /dev/pts/4
```

```
lr-x----- 1 user user 64 Mar 10 19:18 3 -> /proc/93546/fd
```

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

```
total 0
```

```
lrwx----- 1 user user 64 Mar 10 19:18 0 -> /dev/pts/4
```

```
lrwx----- 1 user user 64 Mar 10 19:18 1 -> pipe:[36547]
```

```
lrwx----- 1 user user 64 Mar 10 19:18 2 -> /dev/pts/4
```

```
lr-x----- 1 user user 64 Mar 10 19:18 3 -> /proc/93546/fd
```

# Składnia przekierowań

## Otwarcie pliku do zapisu

*d* > plik (plik zostaje wyzerowany (chyba że `set -o noclobber`, wtedy błąd) lub utworzony)

*d* >| plik (plik zostaje wyzerowany (nawet jeśli jest `set -o noclobber`) lub utworzony)

*d* >> plik (plik zostaje przewinięty na koniec lub utworzony)

## Otwarcie pliku do odczytu bądź zapisu/odczytu

*d* < plik oraz *d* <> plik

## Skopiowanie istniejącego deskryptora (wybór >/< bez znaczenia)

*d* >& *d*' oraz *d* <& *d*'

## Przeniesienie istniejącego deskryptora (wybór >/< bez znaczenia)

*d* >& *d*' - oraz *d* <& *d*' -

## Zamknięcie deskryptora (wybór >/< bez znaczenia)

*d* >& - oraz *d* <& -

**Uwaga:** *d* można pominąć — *d* = 1 dla >, >>, >|, >&; 0 dla <, <& i <> (sic!).

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null — ?`
- ❷ `echo Message 2>/dev/null >&2 —`
- ❸ `echo Message 2>&1 2>/dev/null —`
- ❹ `echo Message 2>&1- 2>/dev/null —`

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

???



# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` —
- ❸ `echo Message 2>&1 2>/dev/null` —
- ❹ `echo Message 2>&1- 2>/dev/null` —

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

Message

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — ?
- ❸ `echo Message 2>&1 2>/dev/null` —
- ❹ `echo Message 2>&1- 2>/dev/null` —

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

???

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — nie ma
- ❸ `echo Message 2>&1 2>/dev/null` —
- ❹ `echo Message 2>&1- 2>/dev/null` —

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

nic nie zostanie wypisane na konsoli

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — nie ma
- ❸ `echo Message 2>&1 2>/dev/null` — ?
- ❹ `echo Message 2>&1- 2>/dev/null` —

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

???

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — nie ma
- ❸ `echo Message 2>&1 2>/dev/null` — jest
- ❹ `echo Message 2>&1- 2>/dev/null` —

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

Message

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — nie ma
- ❸ `echo Message 2>&1 2>/dev/null` — jest
- ❹ `echo Message 2>&1- 2>/dev/null` — ?

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

???

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — nie ma
- ❸ `echo Message 2>&1 2>/dev/null` — jest
- ❹ `echo Message 2>&1- 2>/dev/null` — nie ma

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

nic nie zostanie wypisane na konsoli

# Semantyka przekierowań

- ❶ `echo Message >&2 2>/dev/null` — jest
- ❷ `echo Message 2>/dev/null >&2` — nie ma
- ❸ `echo Message 2>&1 2>/dev/null` — jest
- ❹ `echo Message 2>&1- 2>/dev/null` — nie ma

```
$ echo Message
```

## Kolejność przekierowań

Przekierowania są wykonywane natychmiast („gorliwie”) w kolejności od lewej do prawej.



## Składnia przekierowań (2)

**Przekierowania mogą wystąpić w dowolnym miejscu instrukcji prostej ...**

```
ls /bin >file1
>file1 ls /bin
> file1 ls /bin
ls >file1 /bin
ls > file1 /bin
```

**... lub *po* instrukcji złożonej ...**

```
if true; then echo Yes; fi >file1
```

**... ale nie przed ...**

```
>file1 if true; then echo Yes; fi
bash: syntax error near unexpected token 'then'
```

**... a w środku?**

## Białe znaki

- W odróżnieniu od słów, operatory nie wymagają (ale i nie zabraniają — z wyjątkami!) oddzielania białymi znakami od sąsiadujących tokenów:

```
ls /bin>file1
```

```
ls /bin >file1
```

```
ls /bin > file1
```

```
(>file1 ls)           # dwa operatory obok siebie: ( i >
```

- Operatory przekierowania nie mogą być oddzielone od *poprzedzających* argumentów:

```
ls 2>& 1
```

```
ls 2 >& 1              # 2 jest argumentem ls
```

# Dygresja: wypisywanie na terminal a przekierowania do pliku w libc

## Buforowanie w bibliotece libc(7)

- `fwrite(3)` do `stdout(3)` jest buforowane wierszami, jeśli `stdout` jest terminalem.
- `fwrite` do `stdout` ma wielowierszowy bufor, jeśli `stdout` nie jest terminalem (jeśli tekstu jest mało, to zwykle jest wypisywany jednym syscallem na zakończenie programu).
- `stderr` nie jest buforowany.

## Przekierowania obu strumieni wyjściowych

```
program >file1 2>&1
program 1>file1 2>&1
program &>file1
```

```
program 2>&1 | ...    # przekierowanie wykonywane po połączeniu w potok
program &| ...
```

Jeśli *program* korzysta z libc i wypisuje teksty zarówno do `stdout`, jak i `stderr`, to wiersze pliku `file1` mogą mieć inną kolejność, niż przy wypisywaniu na ekran!

# Dostęp procesu do plików w Linuksie

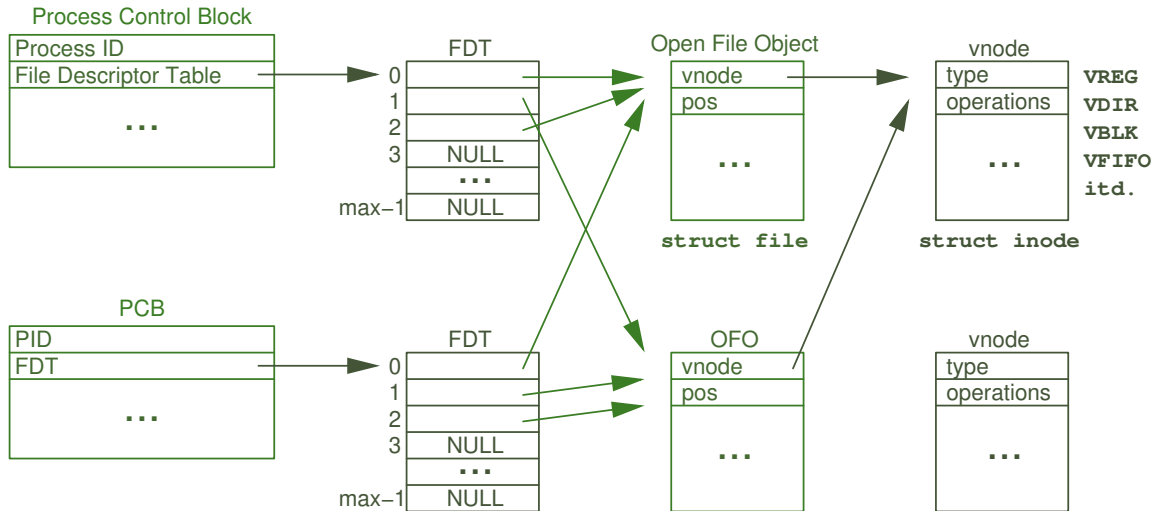
Metadane każdego procesu są przechowywane w strukturze *Process Control Block* (PCB) w *tablicy procesów*, zawierającej m. in.:

- 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 przekaz 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|O_CREAT);  
dup2(k,n);  
close(k);
```

# Implementacja przekierowań

`n > file` (i włączony `noclobber`)

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

# Implementacja przekierowań

`n >| file` (nawet jeśli włączony `noclobber`)

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

```
n >> file
```

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

# Implementacja przekierowań

```
n >> file
```

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

```
n >& m
```

```
dup2(m,n)      Kierunek:  $FDT[n] := FDT[m]$ 
```

# Implementacja przekierowań

```
n >> file
```

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

```
n >& m -
```

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



# Implementacja przekierowań

```
n >> file
```

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

```
n >& -
```

```
close(n)
```

# Implementacja przekierowań

```
n >> file
```

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

```
n >&-      (operatory nie potrzebują spacji)
```

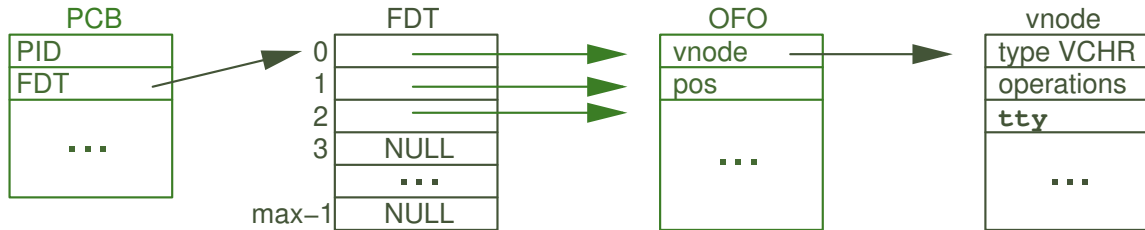
```
close(n)
```

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

Wykonanie konstrukcji `$(...)`:

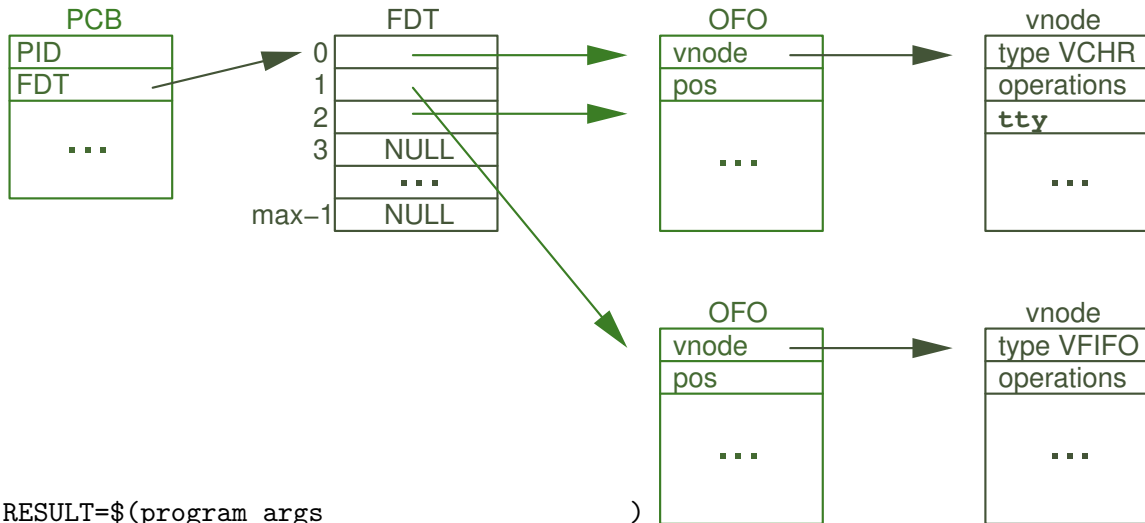
- bash tworzy nowy potok (*fifo*),
- standardowe wyjście `program`-u jest przekierowane do tego potoku,
- bash czyta zawartość tego potoku i tworzy napis (przypisany następnie do zmiennej środowiskowej `RESULT`).

# Zamiana deskryptorów miejscami

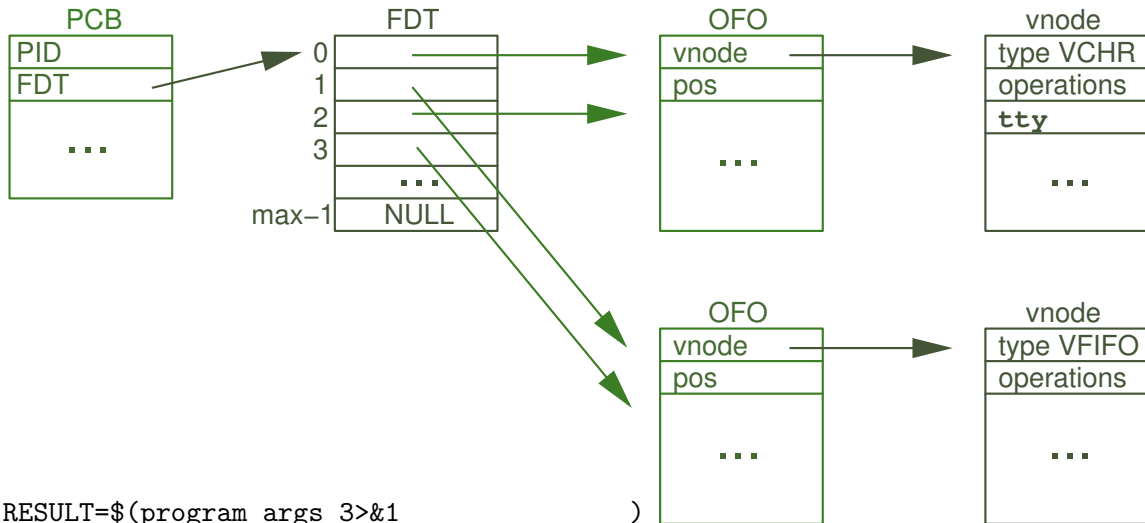


program args

# Zamiana deskryptorów miejscami

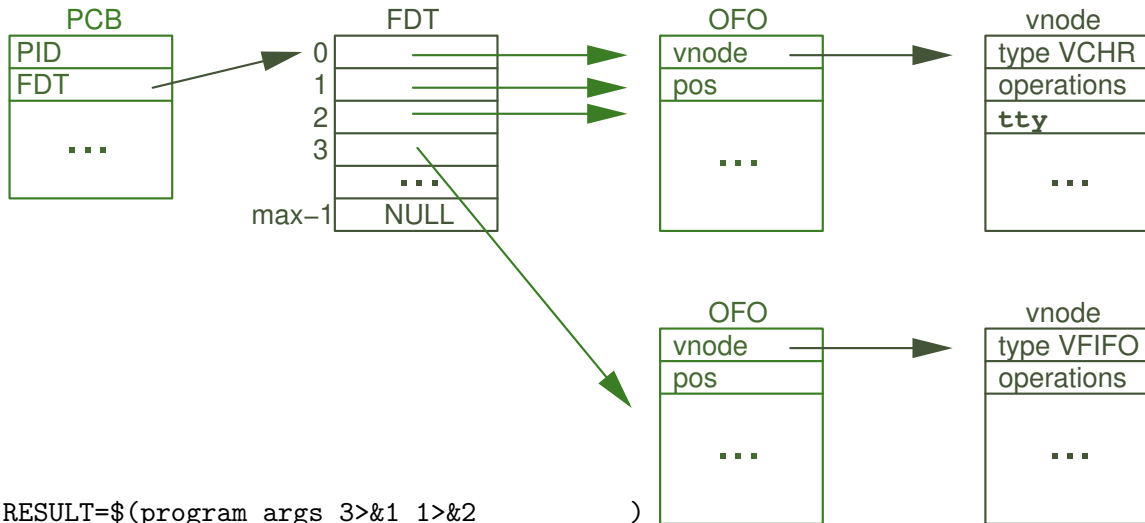


# Zamiana deskryptorów miejscami

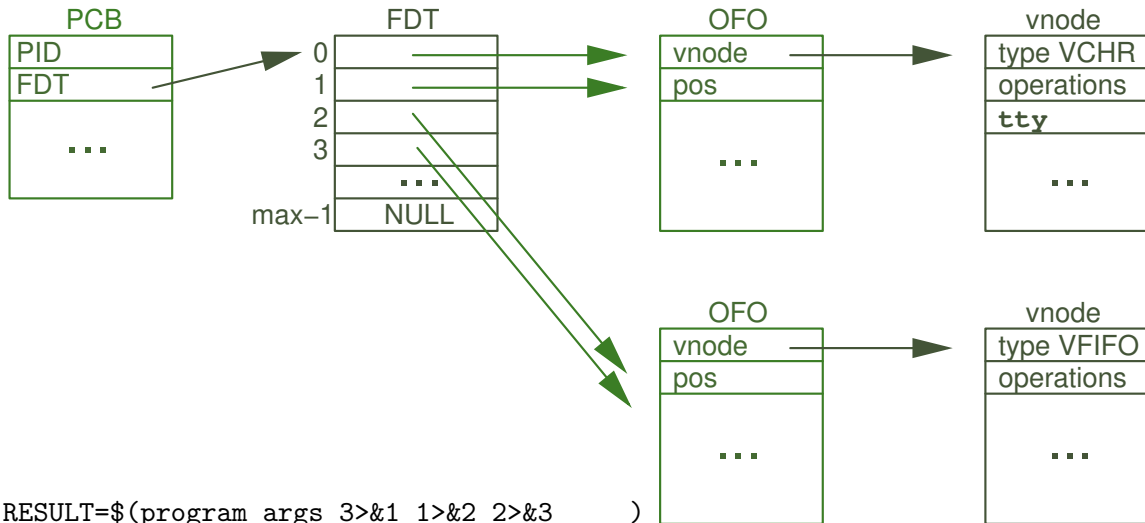


```
RESULT=$(program args 3>&1 )
```

# Zamiana deskryptorów miejscami

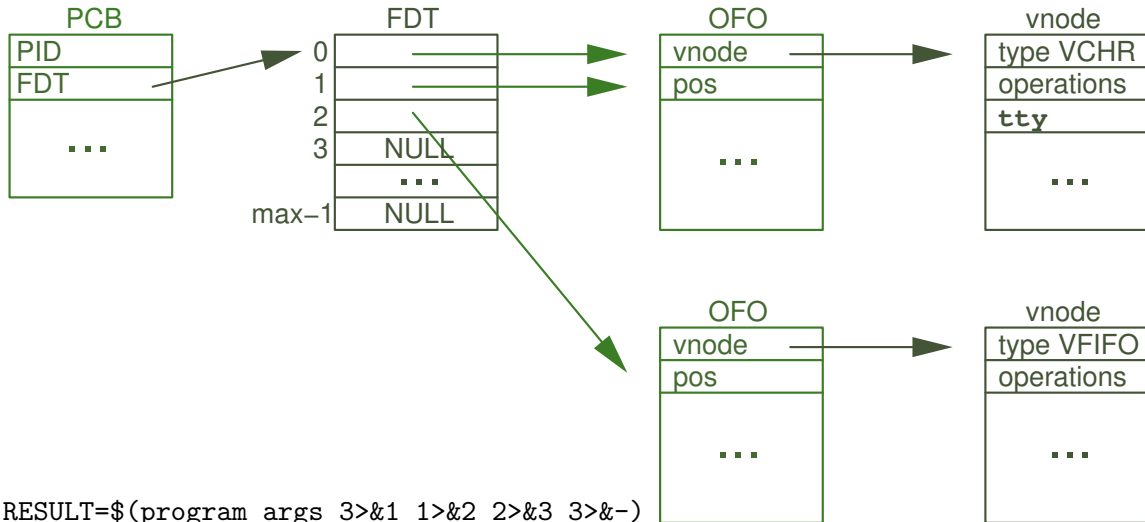


# Zamiana deskryptorów miejscami





# Zamiana deskryptorów miejscami



# Eksperymenty (1)

```
$ echo $(ls -lAF /proc/self/fd/)
total 0 lrwx----- 1 user user 64 Mar 18 04:22 0 -> /dev/pts/1 l-wx----- 1
user user 64 Mar 18 04:22 1 -> pipe:[33264] lrwx----- 1 user user 64 Mar
18 04:22 2 -> /dev/pts/1 lr-x----- 1 user user 64 Mar 18 04:22 3 ->
/proc/7321/fd/
```

Co poszło źle?

# Eksperymenty (1)

```
$ echo $(ls -lAF /proc/self/fd/)
total 0 lrwx----- 1 user user 64 Mar 18 04:22 0 -> /dev/pts/1 l-wx----- 1
user user 64 Mar 18 04:22 1 -> pipe:[33264] lrwx----- 1 user user 64 Mar
18 04:22 2 -> /dev/pts/1 lr-x----- 1 user user 64 Mar 18 04:22 3 ->
/proc/7321/fd/
```

Co poszło źle?

```
$ echo "$(ls -lAF /proc/self/fd/)"
total 0
lrwx----- 1 user user 64 Mar 18 04:22 0 -> /dev/pts/1
l-wx----- 1 user user 64 Mar 18 04:22 1 -> pipe:[33264]
lrwx----- 1 user user 64 Mar 18 04:22 2 -> /dev/pts/1
lr-x----- 1 user user 64 Mar 18 04:22 3 -> /proc/7321/fd/
```

## Eksperymenty (2)

```
$ echo "$(ls -lAF /proc/$$/fd/)"
total 0
lrwx----- 1 user user 64 Mar 18 04:22 0 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 18 04:22 1 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 18 04:22 2 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 18 04:22 255 -> /dev/pts/1
lr-x----- 1 user user 64 Mar 18 04:22 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 18 04:22 0 -> /dev/pts/1
lrwx----- 1 user user 64 Mar 18 04:22 1 -> /dev/pts/1
l-wx----- 1 user user 64 Mar 18 04:22 2 -> pipe:[33390]
l-wx----- 1 user user 64 Mar 18 04:22 3 -> pipe:[33390]
lr-x----- 1 user user 64 Mar 18 04:22 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*: `<<<` (por. rozwinięcia instrukcji `$(...)`).
- Automatyczne tworzenie nowych deskryptorów: `{zmienna}[>|<|>>|...]`.
- `$(< file)` — szybsza wersja `$(cat file)`.

# Here documents

```
cat >&2 << EOT
```

```
Bez serc, bez ducha, - to szkieletów ludy!  
Młodości! dodaj mi skrzydła!  
Niech nad martwym wzlecę światem  
W rajską dziedzinę ułudy:  
Kędy zapał tworzy cudzy,  
Nowości potrząsa kwiatem  
I obleka w nadziei złote malowidła!  
EOT
```

```
function oda {
```

```
    cat >&2 <<- EOT
```

```
    Bez serc, bez ducha, - to szkieletów ludy!  
    Młodości! dodaj mi skrzydła!  
    Niech nad martwym wzlecę światem  
    W rajską dziedzinę ułudy:  
    Kędy zapał tworzy cudzy,  
    Nowości potrząsa kwiatem  
    I obleka w nadziei złote malowidła!  
    EOT
```

```
}
```

Etykieta — dowolne słowo nie występujące w tekście. Zwykle używa się napisu EOT. We wcięciach są dozwolone jedynie znaki tabulacji. Wiersze tekstu nie mogą się wówczas zaczynać znakami tabulacji!

## 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.
- Dobrze w przypadku *logów*, bo współdziela 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 script` — wykonanie skryptu *script* w bieżącej powłoce.
- Jedyny sposób, żeby zmienić środowisko 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
```

## 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...`

## Podjęście 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`).

## Podjęście 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.

## 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`!