# I/O გადამისამართება (Input/Output Redirection) Linux-ში

## სარჩევი



## შესავალი

I/O გადამისამართება არის ძალიან მნიშვნელოვანი კონცეფცია Linux-ში, რომელიც საშუალებას გაძლევთ გააკონტროლოთ, საიდან მოდის ინფორმაცია (შესატანი - input) და სად მიდის შედეგი (გამოსატანი - output). ეს ძალიან გამოსადეგია ყოველდღიურ მუშაობაში და Linux-ის ერთ-ერთი ყველაზე ძლიერი ინსტრუმენტია.

როდესაც ტერმინალში ბრძანებას უშვებთ, ჩვეულებრივ შედეგი ეკრანზე გამოჩნდება და მონაცემებს კლავიატურიდან შეიყვანთ. I/O გადამისამართება საშუალებას გაძლევთ შეცვალოთ ეს ქცევა - შედეგი შეგიძლიათ გააგზავნოთ ფაილში, ხოლო მონაცემები მიიღოთ ფაილიდან ან სხვა პროგრამიდან.

## 1. სამი ძირითადი ნაკადი

Linux-ში ყველა პროგრამას აქვს სამი სტანდარტული ნაკადი:

**STDIN (Standard Input - სტანდარტული შესატანი)** - კოდი: 0

- საიდანაც პროგრამა იღებს მონაცემებს
- ჩვეულებრივ ეს არის კლავიატურა
- პროგრამა ელოდება, რომ მომხმარებელი რაღაცას აკრიფოს

**STDOUT (Standard Output - სტანდარტული გამოსატანი)** - კოდი: 1

- სად გამოჩნდება პროგრამის ნორმალური შედეგი
- ჩვეულებრივ ეს არის ტერმინალის ეკრანი
- აქ ჩნდება წარმატებული ოპერაციების შედეგები

**STDERR (Standard Error - სტანდარტული შეცდომა)** - კოდი: 2

- სად გამოჩნდება შეცდომის შეტყობინებები
- ეს ასევე ჩვეულებრივ ეკრანზე გამოჩნდება
- განცალკევებულია STDOUT-ისგან, რათა შეცდომები ცალკე დამუშავდეს

---

## 2. გამოსატანის გადამისამართება (Output Redirection)

### 2.1. ძირითადი ოპერატორი `>`

ეს ოპერატორი პროგრამის შედეგს აგზავნის ფაილში, ეკრანის ნაცვლად. თუ ფაილი არსებობს, მისი შინაარსი წაიშლება და ჩაიწერება ახალი.

**მაგალითი 1:** ახალი ფაილის შექმნა ტექსტით

```
echo "გამარჯობა სამყარო" > greeting.txt
```

რას აკეთებს ეს ბრძანება:

- `echo` ბრძანება აბეჭდავს ტექსტს "გამარჯობა სამყარო"
- `>` აგზავნის ამ ტექსტს `greeting.txt` ფაილში
- თუ `greeting.txt` უკვე არსებობს, წაშლის მის ძველ შინაარსს

- ეკრანზე არაფერი გამოჩნდება

**მაგალითი 2:** ფაილების სიის შენახვა

```
ls -la > files_list.txt
```

შედეგად `files_list.txt` ფაილში ჩაიწერება მიმდინარე დირექტორიაში არსებული ყველა ფაილის დეტალური ინფორმაცია.

**მაგალითი 3:** სისტემის ინფორმაციის შენახვა

```
date > system_info.txt
uname -a > system_info.txt
```

ამ შემთხვევაში მხოლოდ `uname -a`-ს შედეგი შეინახება, რადგან მეორე `>` პირველს გადააწერს.

## 2.2. დამატების ოპერატორი `>>`

ეს ოპერატორი არ შლის ძველ ინფორმაციას, არამედ ამატებს ახალს ფაილის ბოლოში.

**მაგალითი:**

```
echo "პირველი ხაზი" > myfile.txt
echo "მეორე ხაზი" >> myfile.txt
echo "მესამე ხაზი" >> myfile.txt
```

შედეგი `myfile.txt` ფაილში:

```
პირველი ხაზი
მეორე ხაზი
მესამე ხაზი
```

**პრაქტიკული გამოყენება - ლოგების შენახვა:**

```
echo "2026-01-19: სისტემა გაეშვა" >> system_log.txt
echo "2026-01-19: მომხმარებელი შევიდა" >> system_log.txt
```

## 2.3. შეცდომების გადამისამართება `2>`

ეს ოპერატორი გადამისამართებს მხოლოდ შეცდომის შეტყობინებებს (STDERR).

**მაგალითი:**

```
ls /arsebuli_ara 2> errors.txt
```

რას აკეთებს:

- `ls` ცდილობს ნახოს `/arsebuli_ara` დირექტორია
- თუ ეს დირექტორია არ არსებობს, შეცდომის შეტყობინება ჩაიწერება `errors.txt` ფაილში
- ეკრანზე არაფერი გამოჩნდება (არც შეცდომა)

**მაგალითი - შეცდომების იგნორირება:**

```
find / -name "myfile.txt" 2> /dev/null
```

ეს ბრძანება მოძებნის `myfile.txt` ფაილს მთელ სისტემაში, მაგრამ შეცდომები (როგორიცაა "Permission denied") არ გამოჩნდება.

## 2.4. ყველაფრის გადამისამართება ერთ ფაილში

**მაგალითი:**

```
ls -la /home /arsebuli_ara > output.txt 2>&1
```

რას აკეთებს:

- `ls -la /home /arsebuli_ara` ცდილობს ნახოს ორივე დირექტორია
- `> output.txt` - ნორმალური შედეგი ( `/home` -ის შიგთავსი) მიდის ფაილში
- `2>&1` - შეცდომები (არასებული დირექტორიის შესახებ) ასევე მიდის იქ, სადაც ნორმალური შედეგი
- ერთ ფაილში ყველაფერია

**თანამედროვე სინტაქსი (bash 4+):**

```
ls -la /home /arsebuli_ara &> output.txt
```

ეს იგივეა, რაც `> output.txt 2>&1` , უბრალოდ უფრო მოკლე.

## 3. შესატანის გადამისამართება (Input Redirection)

### 3.1. ძირითადი ოპერატორი `<`

ეს ოპერატორი პროგრამას აძლევს ინფორმაციას ფაილიდან, კლავიატურის ნაცვლად.

**მაგალითი 1:** ხაზების დათვლა

```
wc -l < myfile.txt
```

რას აკეთებს:

- `wc -l` ითვლის ხაზების რაოდენობას
- `< myfile.txt` აწვდის `myfile.txt` ფაილის შინაარსს პროგრამას
- შედეგი: რიცხვი, რამდენი ხაზია ფაილში

**მაგალითი 2:** ტექსტის დალაგება

```
sort < names.txt
```

დაალაგებს `names.txt` ფაილში ჩაწერილ სახელებს ანბანის მიხედვით და დაბეჭდავს ეკრანზე.

**განსხვავება არგუმენტსა და გადამისამართებას შორის:**

```
wc -l myfile.txt      # აჩვენებს: "10 myfile.txt"
wc -l < myfile.txt    # აჩვენებს: "10"
```

### 3.2. Here Document `<<`

ეს საშუალებას გაძლევთ დაწეროთ მრავალხაზიანი ტექსტი პირდაპირ ბრძანების შიგნით.

**მაგალითი:**

```
cat << EOF > info.txt
ეს არის პირველი ხაზი
ეს არის მეორე ხაზი
ეს არის მესამე ხაზი
EOF
```

რას აკეთებს:

- `cat` კითხულობს ტექსტს `<<` და `EOF` შორის
- `> info.txt` წერს ამ ტექსტს ფაილში
- `EOF` არის დამასრულებელი მარკერი (შეგიძლიათ გამოიყენოთ ნებისმიერი სიტყვა)

**პრაქტიკული გამოყენება - კონფიგურაციის ფაილის შექმნა:**

```
cat << END > config.conf
server_name=myserver
port=8080
debug=true
END
```

---

# 4. Pipe (მილი) `|` - პროგრამების დაკავშირება

## 4.1. რა არის Pipe?

Pipe ( `|` ) აკავშირებს ერთი ბრძანების **გამოსატანს** (output) მეორე ბრძანების **შესატანთან** (input). ეს Linux-ის ერთ-ერთი ყველაზე ძლიერი თვისებაა - მარტივი ბრძანებების კომბინირებით რთული ამოცანების გადაწყვეტა.

**ძირითადი იდეა:** პირველი ბრძანება აწარმოებს output-ს → ეს output ხდება მეორე ბრძანების input → მეორე ბრძანება ამუშავებს და აწარმოებს საბოლოო შედეგს.

## 4.2. პრაქტიკული მაგალითები

**მაგალითი 1:** ფაილების დათვლა

```
ls -l | wc -l
```

რას აკეთებს ნაბიჯ-ნაბიჯ:

1. `ls -l` აბრუნებს ფაილების სიას (თითო ფაილი ცალკე ხაზზე)
2. `|` ამ სიას აგზავნის `wc` ბრძანებაში
3. `wc -l` ითვლის ხაზების რაოდენობას
4. შედეგი: რამდენი ფაილია დირექტორიაში

**მაგალითი 2:** ტექსტის ძებნა და დალაგება

```
cat names.txt | grep "a" | sort
```

რას აკეთებს:

1. `cat names.txt` კითხულობს ფაილს
2. `grep "a"` ტოვებს მხოლოდ იმ ხაზებს, სადაც არის ასო "a"
3. `sort` ალაგებს ამ შედეგებს ანბანის მიხედვით
4. დალაგებული სია ეკრანზე გამოჩნდება

**მაგალითი 3:** პროცესების ძებნა

```
ps aux | grep firefox
```

აჩვენებს ყველა firefox-თან დაკავშირებულ პროცესს.

**მაგალითი 4:** უნიკალური სიტყვების დათვლა

```
cat text.txt | tr ' ' '\n' | sort | uniq -c
```

რას აკეთებს:

1. `cat text.txt` - კითხულობს ფაილს
2. `tr ' ' '\n'` - შლის ყველა სპეისს და ცვლის ახალი ხაზით (თითო სიტყვა ცალკე ხაზზე)
3. `sort` - ალაგებს სიტყვებს
4. `uniq -c` - ითვლის, რამდენჯერ მეორდება თითოეული სიტყვა

**მაგალითი 5:** ყველაზე დიდი ფაილების მოძებნა

```
ls -lh | sort -k5 -hr | head -5
```

აჩვენებს 5 ყველაზე დიდ ფაილს მიმდინარე დირექტორიაში.

**მაგალითი 6:** IP მისამართების გაფილტვრა

```
cat access.log | grep "404" | awk '{print $1}' | sort | uniq
```

ლოგ ფაილიდან იღებს ყველა უნიკალურ IP მისამართს, რომელმაც მიიღო 404 შეცდომა.

---

## 5. მთავარი განსხვავება: `>` vs `|`

### 5.1. გადამისამართება `>`

- **მიზანი:** ფაილი
- **რას აკეთებს:** იღებს პროგრამის output-ს და წერს ფაილში
- **გამოყენება:** როცა გინდა შედეგის შენახვა ფაილში მომავალი გამოყენებისთვის
- **მაგალითი:** `ls > list.txt` - ქმნის ფაილს სიით
- **შედეგი:** ინფორმაცია ფაილშია, ეკრანზე არაფერი გამოჩნდება

### 5.2. Pipe `|`

- **მიზანი:** სხვა პროგრამა/ბრძანება
- **რას აკეთებს:** იღებს ერთი პროგრამის output-ს და აგზავნის მეორე პროგრამის input-ში
- **გამოყენება:** როცა გინდა შედეგის დამუშავება სხვა ბრძანებით
- **მაგალითი:** `ls | grep ".txt"` - ფილტრავს შედეგებს
- **შედეგი:** გადამუშავებული ინფორმაცია ეკრანზე ან შემდეგ ბრძანებაში გადადის

### 5.3. გამჭვირვალე მაგალითი

**`>` - ფაილში ჩაწერა:**

```
echo "გამარჯობა" > file.txt
```

- შედეგი: "გამარჯობა" ჩაიწერა `file.txt` ფაილში
- ეკრანზე: არაფერი
- `file.txt` ფაილში: "გამარჯობა"

**`|` - პროგრამაში გადაცემა:**

```
echo "გამარჯობა" | wc -c
```

- შედეგი: `wc -c` ითვლის რამდენი სიმბოლოა "გამარჯობა"-ში
- ეკრანზე: რიცხვი (მაგალითად, 11)
- არსად არ ინახება, უბრალოდ დამუშავდა და გამოჩნდა

**კომბინირებული გამოყენება:**

```
ls -l | grep ".txt" > text_files.txt
```

- `ls -l` აბრუნებს ფაილების სიას
- `|` აგზავნის `grep`-ში
- `grep ".txt"` ტოვებს მხოლოდ .txt ფაილებს
- `>` წერს ამ გაფილტრულ სიას `text_files.txt` ფაილში

**მარტივი წესი:**

- თუ შედეგი **ფაილში** გინდა → გამოიყენე `>`
- თუ შედეგი **სხვა პროგრამას** უნდა გადასცე → გამოიყენე `|`

---

## 6. გავრცელებული შეცდომები და მათი თავიდან აცილება

**შეცდომა 1: `>` და `>>`-ის აღრევა**

ცუდი მაგალითი:

```
echo "ხაზი 1" > log.txt
echo "ხაზი 2" > log.txt  # ხაზი 1 წაიშალა!
```

შედეგი: `log.txt` ფაილში მხოლოდ "ხაზი 2" იქნება.

კარგი მაგალითი:

```
echo "ხაზი 1" > log.txt
echo "ხაზი 2" >> log.txt  # დაემატა არსებულს
```

შედეგი: `log.txt` ფაილში ორივე ხაზი იქნება.

**როგორ ავირიდოთ:** ჯერ ფიქრი, რომ არსებული ინფორმაცია უნდა შენარჩუნდეს თუ არა. თუ კი → გამოიყენე `>>`.

---

### შეცდომა 2: Pipe-ის გამოყენება ფაილისთვის

ცუდი მაგალითი:

```
cat file.txt | output.txt  # არ იმუშავებს!
```

ეს გამოიწვევს შეცდომას, რადგან `output.txt` არ არის ბრძანება.

სწორი ვარიანტი:

```
cat file.txt > output.txt
```

**როგორ ავირიდოთ:** გახსოვდეს - `|` მხოლოდ ბრძანებებისთვის, `>` ფაილებისთვის.

---

### შეცდომა 3: გადამისამართების მიმდევრობა

ცუდი მაგალითი:

```
> output.txt ls -la
```

მუშაობს, მაგრამ დამაბნეველია და ცუდი სტილია.

კარგი მაგალითი:

```
ls -la > output.txt
```

გადამისამართება ბოლოში უფრო გასაგებია.

---

### შეცდომა 4: არსებული ფაილის გადაწერა შემთხვევით

საშიში:

```
sort file.txt > file.txt  # ფაილი დაზიანდება!
```

ეს შემთხვევაში გაასუფთავებს `file.txt` ფაილს, სანამ `sort` მოასწრებს მის წაკითხვას.

სწორი გზა:

```
sort file.txt > file_sorted.txt
mv file_sorted.txt file.txt
```

ან უფრო მოკლედ:

```
sort file.txt -o file.txt
```

---

### შეცდომა 5: STDOUT და STDERR-ის გადამისამართების მიმდევრობა

ცუდი მაგალითი:

```
command 2>&1 > output.txt  # არ მუშაობს როგორც ველოდებით!
```

ამ შემთხვევაში მხოლოდ STDOUT მიდის ფაილში, STDERR კი ეკრანზე დარჩება.

სწორი მიმდევრობა:

```
command > output.txt 2>&1
```

ან უფრო მარტივად:

```
command &> output.txt
```

**როგორ ავირიდოთ:** ჯერ `>` შემდეგ `2>&1`, ან გამოიყენე `&>`.

---

## სასარგებლო რჩევები

### რჩევა 1: მნიშვნელოვანი ფაილების დაზღვევა

გადაწერამდე ყოველთვის შექმენით ბექაპი:

```
cp important.txt important.txt.backup
echo "ახალი ტექსტი" > important.txt
```

ან გამოიყენეთ `>>` ახლის დამატებისთვის ძველის წაშლის გარეშე.

---

### რჩევა 2: ყველაფრის იგნორირება /dev/null-ით

თუ არც output და არც შეცდომები არ გინდათ:

```
command > /dev/null 2>&1
```

`/dev/null` არის "შავი ხვრელი" - ყველაფერი რაც მას გააგზავნით, უბრალოდ ქრება.

პრაქტიკული მაგალითი:

```
find / -name "myfile.txt" 2> /dev/null
```

მოძებნის ფაილს მთელ სისტემაში, მაგრამ "Permission denied" შეცდომები არ გამოჩნდება.

---

### რჩევა 3: ფაილის არსებობის შემოწმება

`>`-ით გადაწერამდე შეამოწმეთ არსებობს თუ არა:

```
if [ -f output.txt ]; then
    echo "ფაილი უკვე არსებობს!"
else
    ls > output.txt
fi
```

ან ერთ ხაზად:

```
[ -f output.txt ] && echo "ფაილი არსებობს!" || ls > output.txt
```

---

### რჩევა 4: ერთდროულად ეკრანზე და ფაილში - `tee`

თუ გინდათ, რომ output ეკრანზეც გამოჩნდეს და ფაილშიც ჩაიწეროს:

```
ls -la | tee files.txt
```

რას აკეთებს `tee`:

- იღებს input-ს pipe-ით
- წერს ფაილში
- ბეჭდავს ეკრანზეც

დამატებით ( `>>` ანალოგი):

```
ls -la | tee -a files.txt
```

---

### რჩევა 5: რამდენიმე ბრძანების output ერთ ფაილში

```
{
    echo "=== სისტემის ინფორმაცია ==="
    date
    echo "=== მომხმარებლები ==="
    who
    echo "=== დისკის სივრცე ==="
    df -h
} > system_report.txt
```

---

### რჩევა 6: Pipe-ების დაკოპირება და რედაქტირება

რთული pipe chain-ები წერისას, შეგიძლიათ დაშალოთ ნაწილებად:

```
# ჯერ პირველი ნაბიჯი
cat file.txt > temp1.txt

# შემდეგ მეორე
grep "pattern" temp1.txt > temp2.txt

# შემდეგ ბოლო
sort temp2.txt > result.txt

# როცა დარწმუნდებით, შეკრიბეთ ერთად:
cat file.txt | grep "pattern" | sort > result.txt
```

---

**რჩევა 7: გრძელი pipe-ების გაშლა რამდენიმე ხაზზე**

კითხვადობისთვის:

```
cat access.log \
    | grep "ERROR" \
    | awk '{print $1, $7}' \
    | sort \
    | uniq -c \
    | sort -nr \
    | head -10 > top_errors.txt
```

`\` ბრძანების გაგრძელებაა შემდეგ ხაზზე.

---

# შეჯამება

I/O გადამისამართება და Pipe-ები არის Linux-ის ფუნდამენტური ინსტრუმენტები, რომლებიც საშუალებას გაძლევენ:

**გადამისამართების შესაძლებლობები:**

- შეინახოთ ბრძანების შედეგები ფაილებში ( `>` , `>>` )
- წაიკითხოთ მონაცემები ფაილებიდან ( `<` )
- დაამუშაოთ შეცდომები ცალკე ( `2>` )
- გააერთიანოთ output და error ერთ ფაილში ( `2>&1` , `&>` )

**Pipe-ის შესაძლებლობები:**

- გააერთიანოთ მარტივი ბრძანებები რთულ ამოცანებში (`|`)
- შექმნათ მონაცემთა დამუშავების კონვეიერები
- დააკომბინიროთ სპეციალიზებული ინსტრუმენტები