Systemy operacyjne

Laboratorium 3

Find

Jarosław Rudy
Politechnika Wrocławska

28 lutego 2017

Laboratorium obejmuje wykorzystanie i zrozumienie komendy `find` wraz ze znajomością systemu plików i uprawnień do plików.

1 Ograniczenia

- Zakaz wywoływania komendy `find` więcej niż raz w skrypcie (chyba, że zadanie lub prowadzący wprost na to zezwoli).
- Skrypt, poza standardowymi elementami (jak sprawdzenie argumentów) nie powinien wykorzystywać komend innych niż `find`.
- Ścieżki i wyrażenie `find`a należy podawać jawnie tzn. zakazane jest wywołanie `find` zamiast `find . -print`.

2 Podstawy

Wbrew nazwie i jej najczęściej spotykanemu użyciu, komenda `find` nie jest ograniczona do znajdywania plików i wypisywania ich nazw na ekran. Sposób działania komendy `find` jest dużo bardziej elastyczny i może zostać podsumowany w następujący sposób:

Komenda `find` określa zbiór plików,
a następnie przetwarza każdy plik z tego zbioru
zgodnie z podanym *wyrażeniem*.

Istnieje możliwość wywołania komendy `find` bez żadnych argumentów (tak naprawdę odpowiada to wywołaniu `find . -print`), jednakże zadania laboratoryjne zwykle wymagają podania dla komendy wielu argumentów. Argumenty te można podzielić na 3 podstawowe grupy: opcje podstawowe, ścieżki poszukiwań i wyrażenie testowe.

1. Podstawowe opcje powinny zostać podane przed ścieżkami poszukiwań. Ich główne wykorzystanie to kontrola sposobu traktowania dowiązań symbolicznych. Do opcji takich należą `-P` (nie podążaj za symlinkami) oraz `-L` (podążaj za symlinkami). Domyślnym zachowaniem jest nie podążanie za symlinkami. W takim przypadku jeśli testowany plik jest symlinkiem, to testowane będą właściwości samego symlinka. Jeśli podążamy za linkami, to testowane będą właściwości celu symlinka, a nie sam symlink (chyba, że symlink jest zepsuty). Ponadto, jeśli przetwarzany symlink wskazuje na katalog, a my podążamy za linkami, to zawartość wskazywanego katalogu (drzewa katalogowego) zostanie również "przeszukana", nawet jeśli prowadzi w zupełnie inne miejsce systemu plików. Innymi słowy, jeśli `link` jest symlinkiem na katalog, to wywołania:

   ```
   find -P link -print
   find -L link -print
   ```

   będą miały zupełnie inne działanie. Pierwsze przetworzy tylko plik `link`, podczas gdy drugie przetworzy także wszystkie pliki w drzewie katalogowym wskazywanym przez `link`.

2. Ścieżki poszukiwań określają, które pliki lub drzewa katalogowe należy przeszukać (przetworzyć). Po pierwsze, można podać więcej niż jedną ścieżkę. Po drugie ścieżką nie musi być katalog – zwykły plik też zostanie przetworzony. Zwykle jednak podajemy w tym miejscu katalogi, które traktowane są jako *drzewa katalogowe*. Przez drzewo takie rozumiemy wszystkie pliki, które znajdują się "poniżej" danego katalogu. Przykładowo, wszystkie poniższe pliki:

   ```
   katalog
   katalog/plik
   katalog/podkatalog
   katalog/podkatalog/plik2
   katalog/podkatalog/podpodkatalog
   katalog/podkatalog/podpodkatalog/plik3
   ```

   należą do drzewa katalogowego, którego korzeniem jest `katalog`. Komenda `find` domyślnie bierze pod uwagę wszystkie pliki w danym drzewie katalogowym, nieważne jak głęboko są zagnieżdżone. W celach diagnostycznych można w miarę wygodnie wyświetlać drzewa katalogowe przy wykorzystaniu komendy `tree`.

3. Wyrażenie testowe (lub po prostu wyrażenie) jest najważniejszą i najbardziej skomplikowaną częścią komendy `find` i zostanie omówione w dalszej części instrukcji.

3 Wyrażenie

Wyrażenie określa, jakie właściwości plików należy zbadać i jakie czynności należy podjąć. Wyrażenie ma charakter logiczny i, podobnie do wyrażeń z języka `C/C++`, składa się z mniejszych wyrażeń (elementów) połączonych przy pomocy operatorów logicznych. Przykładowo, jeśli `elem1` i `elem2` zostaną połączone operatorem typu `AND`, to `elem2` nie zostanie wykonany, jeśli `elem1` dał wynik `false` (bo wartość logiczna całego wyrażenia jest już znana). Operatory są następujące:

- Koniunkcja logiczna (AND): `-and`, `-a` lub brak operatora. Tak więc poniższe trzy formy:

  ```
  elem1 -and elem2 -and elem3
  elem1 -a elem2 -a elem3
  elem1 elem2 elem3
  ```

  są równoważne.

- Alternatywa logiczna (OR): `-or` lub `-o`.

- Negacja logiczna (NOT): `-not` lub wykrzyknik (`!`).

- Nawias: nawiasy okrągłe `"("` oraz `")"`. Nawiasy są znakami specjalnymi `bash`a, więc należy je wyescape'ować (z użyciem apostrofów lub odwrotnego ukośnika).

- Przecinek. Działa jak analogiczny operator z `C/C++` tzn. w poniższym przykładzie:

  ```
  elem1 , elem2 , elem3
  ```

  wykonane zostaną *wszystkie* trzy elementy, niezależnie od ich wartości logicznej, zaś wartością logiczną całości jest najbardziej na prawo wysunięty element (w tym przypadku `elem3`).

Zauważmy, że wszystkie operatory są argumentami dla `find`a, wobec czego muszą wystąpić spacje oddzielające je od innych argumentów (opcji ogólnych, ścieżek, operatorów, elementów).

Po odliczeniu operatorów, pozostała część wyrażenia składa się z elementów, które mogą należeć do trzech typów:

- Testy. Nigdy nie powodują wystąpenia efektów ubocznych (np. zmian w systemie, wyświetlania tekstu), zaś wartość logiczna testu jest zależna od testowanego pliku. Przykładowo test `-links 2` sprawdza czy przetwarzany plik ma *dokładnie* dwa dowiązania *twarde*.

- Akcje. Wartość logiczna akcji jest z reguły ustalona (np. `-print` zwraca `true`, zaś `-false` oczywiście zwraca `false`). Ponadto, akcje mają efekty uboczne, typu wywołanie komendy (`-exec`), wyświetlenie tekstu (`-print`, `-printf` i podobne) czy usunięcie pliku (`-delete`).

- Opcje. Nie należy ich mylić z opcjami podstawowymi (typu `-L` i `-P`) omówionymi wcześniej. Zwracają zawsze prawdę. Ich istotną cechą jest to, że prawie wszystkie z nich działają globalnie, niezależnie od położenia opcji w wyrażeniu. Przykładowo, poniższe wywołania:

  ```
  find . -mindepth 2 -print
  find . -print -mindepth 2
  ```

  są równoważne. Z tego względu opcje wygodnie jest umieszczać na początku wyrażenia, dla uniknięcia nieporozumień.

Zauważmy, że opcja `-mindepth` wymaga argumentu i musi być on podany *po spacji*! Działa tak bardzo wiele testów czy akcji. Przykładowo `-type f` testuje czy plik jest plikiem zwykłym, zaś `-type d` testuje czy plik jest katalogiem. Niektóre testy/akcje nie mają argumentu (np. `-print`), a niektóre mogą mieć bardzo skomplikowany argument (np. `-printf`). Warto też zwrócić uwagę na różne wersje testów/akcji: `-type` i `-ltype`, `-name` i `-iname` itp.

Niektóre testy przyjmujące argument liczbowy mogą dodatkowo posiadać przed nim znak plus lub minus. Z poniższych testów:

```
-size 2
-size -2
-size +2
```

pierwszy sprawdza czy przetwarzany plik ma dwie jednostki długości, drugi czy ma mniej niż dwie jednostki, a trzeci czy ma więcej niż dwie jednostki. Zauważmy też, że domyślną jednostką dla `-size` są *bloki* a nie znaki/bajty! Jednostkę można zmienić dodając odpowiedni znak po liczbie np. `k` dla kibibajtów (kiB)[1].

4 Praktyczne użycie `find`

W przypadku oryginalnej komendy `find` nie będzie żadnego widocznego efektu, jeśli nie podana zostanie chociaż jedna akcja. Tak więc komenda:

```
find DIR -size 0
```

nie wypisze na ekran nic, nawet jeśli katalog `DIR` zawiera puste pliki. Jednakże współczesna implementacja `find` automatycznie doda akcję `-print` na końcu wyrażenia, ale tylko jeśli w wyrażeniu nie ma żadnych innych akcji lub jedyną akcją jest `-prune`. Ponadto, jeśli nie podano żadnych ścieżek poszukiwań, to automatycznie użyta zostanie ścieżka aktualnego katalogu. Tak więc wywołania:

[1] Nie mylić z kilobajtami (kB)! 1 kiB = 1.024 kB = 1024 B.

```
find
find -size 0 -prune
```

zostaną zrozumiane jako:

```
find . -print
find . -size 0 -prune -print
```

Ponadto należy pamiętać, że w wyrażeniu może występować wiele akcji i to w różnych miejscach wyrażenia. Weźmy za przykład wywołanie:

```
find DIR -print -type d -print
```

Pierwsza akcja `-print` zadziała dla każdego pliku w katalogu `DIR`, więc nazwy tych plików zostaną wypisane na ekran. Jednakże, druga akcja `-print` zadziała jedynie dla tych plików, które są katalogami. Jeśli plik nie jest katalogiem, to test `-type d` zwróci fałsz, co spowoduje, że całe wyrażenie ma już znaną wartość logiczną (pamiętamy, że domyślnym operatorem łączącym opcje jest `AND`), więc dalsza część wyrażenia zostanie pominięta.

Na zakończenie tej sekcji należy zwrócić uwagę na bardzo ważną akcję `-printf` (nie mylić z `-print`!), która pozwala drukować dowolny tekst, a także różne informacje o przetwarzanym pliku, takie jak nazwa pliku , ścieżka, poziom zagłębienia w drzewie katalogowym itp.

5 Uprawnienia plików

W tej części zajmiemy się uprawnieniami plików w systemach linuksowych oraz ich zastosowaniem dla komendy `find`. Podstawowe uprawnienia plikowe zapisane są z użyciem 9 bitów, podzielonych na 3 grupy. W każdej grupie znajdują się 3 bity odpowiedzialne za prawa odczytu (bit Read, r), zapisu/modyfikacji (bit Write, w) oraz wykonania (but eXecute, x). Jeśli bit jest ustawiony (wartość 1), to prawo jest przyznane, jeśli zaś bit jest wyczyszczony (0), to prawo nie jest przyznane. Prawa plików można sprawdzić z użyciem komendy `stat` lub `ls -l`[2]. Prawa przyznane oznaczane są jako `'r'`, `'w'` lub `'x'`, zaś prawa nie przyznane oznaczane są przez minus (`'-'`). Znaczenie praw r, w oraz x jest oczywiste dla plików zwykłych, zaś nieco inne dla katalogów. Linki symboliczne zawsze mają pełne prawa (tj. `rwxrwxrwx`).

Trzy osobne grupy praw wynikają z istnienia trzech możliwych sytuacji dostępu do pliku. Zacznijmy od tego, że każdy plik posiada przypisane dwa numery:

- identyfikator użytkownika (`user id`, `uid`). Oznacza właściciela pliku, nazywanego też po prostu właścicielem (ewentualnie użytkownikiem),
- identyfikator grupy (`group id`, `gid`). Oznacza właściciela grupowego, zwanego też po prostu grupą.

[2] `ls -l` wypisuje na początku praw dodatkowy znak określający typ pliku (`'-'` dla plików zwykłych, `'d'` dla katalogów itp.).

Podczas każdej próby odczytu/zapisu/wykonania pliku system musi sprawdzić czy aktualny proces posiada odpowiednie uprawnienia. Po pierwsze, należy określić `uid` oraz `gid` danego procesu. Zwykle jest to po prostu `uid`/`gid` użytkownika systemu, który uruchomił dany proces. Użytkownik posiada tylko jeden `uid`, ale może posiadać więcej niż jeden `gid` (tzn. może należeć do wielu grup).

Mając `uid`/`gid` możemy przejść do sprawdzania praw. Sprawdzane są trzy następujące przypadki w *dokładnie* tej kolejności:

1. Jeśli `uid` procesu pasuje do `uid` pliku, to jesteśmy traktowani jak właściciel pliku (`user`, `u`).
2. W przeciwnym przypadku, jeśli co najmniej jeden `gid` procesu pasuje do `gid` pliku, to jesteśmy traktowani jako właściciel grupowy pliku (`group`, `g`).
3. W przeciwnym przypadku jesteśmy traktowani jako "inny" (`other`, `o`).

Dlatego istnieje 9 bitów uprawnień: bity 1–3 przeznaczone są dla właściciela, bity 4–6 dla właściciela grupowego, a bity 7–9 dla pozostałych tzn.:

```
rwxrwxrwx
uuugggooo
```

Należy podkreślić, że jeśli jesteśmy zarówno właścicielem jak i właścicielem grupowym, to prawa właściciela mają priorytet. Innymi słowy, jeśli prawa właściciela są odebrane, to nie otrzymamy dostępu do pliku, nawet jeśli prawa grupy są nam przyznane. Można to podsumować w następujący sposób:

1. Jeśli jesteśmy tylko właścicielem, liczą się bity 1–3.
2. Jeśli jesteśmy tylko właścicielem grupowym, liczą się bity 4–6.
3. Jeśli jesteśmy zarówno właścicielem jak i właścicielem grupowym, liczą się bity 1–3.
4. Jeśli nie jesteśmy ani właścicielem, ani właścicielem grupowym, liczą się bity 7–9.

Przejdźmy teraz do praktyki. Ogólnie występują dwa typy pytań, które możemy zadać:

1. Czy dany użytkownik/proces może uzyskać dostęp (np. odczyt) do danego pliku?
2. Czy dany plik przyznaje dane prawo (np. odczyt) dla danego "podmiotu" (właściciel, właściciel grupowy lub pozostali)?

Odpowiedź na pierwsze pytanie jest prosta. Aby sprawdzić czy aktualny użytkownik/proces będzie mógł otworzyć dany plik można posłużyć się komenda `test` z opcją `-r` lub wykorzystać test `-readable` komendy `find`. Reszta dzieje

się automatycznie (system operacyjny sam określi nasz `uid`/`gid`, sprawdzi, który przypadek zachodzi i wypracuje odpowiedź). Dla praw zapisu i wykonania sytuacja jest analogiczna.

Drugie pytanie jest dużo trudniejsze, gdyż wymaga analizy konkretnych bitów praw pliku. Dla uproszczenia załóżmy konkretny przypadek: chcemy znaleźć (wydrukować nazwy) plików, które przyznają prawo odczytu właścicielowi (`u`), ale NIE przyznają prawa zapisu właścicielowi grupowemu (`g`). Pliki, które spełniają te warunki będą miały prawa postaci:

```
r***-****
```

lub w formie maski bitowej:

```
1***0****
```

przy czym gwiazdka oznacza, że dany bit nie jest istotny (może być zerem albo jedynką).

Jak sprawdzać tego typu rzeczy przy użyciu `find`a? Wykorzystywane są do tego trzy testy (opcje) z "rodziny" `-perm`. Cechą wspólną wszystkich trzech jest podanie maski bitowej w formie ósemkowej. Przykładowo, maska 110100100 musi być zapisana jako 3 cyfry ósemkowe, czyli 644. Są to bardzo typowe prawa dla zwykłych plików, jako że przyznają odczyt/zapis dla właściciela pliku oraz odczyt dla grupy i innych. Maskę można także podać w formie symbolicznej, ale nie jest to zalecane.

Trzy wspomniane testy `-perm` różnią się, oprócz formy, sposobem interpretacji maski.

1. `-perm mask`. Test ten traktuje maskę w sposób dosłowny. Jeśli bit w masce ma wartość 1, to dane prawo MUSI być przyznane, żeby test zwrócił wartość `TRUE`, jeśli zaś bit w masce ma wartość 0, to dane prawo NIE MOŻE być przyznane. Oznacza to, że tylko 1 z $2^9 = 512$ ciągów 9 bitów sprawi, że ta wersja `-perm` zwróci wartość `TRUE`. W naszym przykładzie (`1** *0* ***`) mamy 128 pasujących ciągów, więc musielibyśmy wykorzystać 128 opcji `-perm` połączonych operatorem `OR`. Oczywiście nie jest to dobry pomysł.

2. `-perm -mask`. Jeśli bit w masce ma wartość 1, to dane prawo w pliku MUSI być przyznane. Reszta bitów w prawach pliku NIE ma znaczenia. Innymi słowy, wszystkie jedynki z maski muszą być przyznanymi prawami jednocześnie (na zasadzie operatora `AND`).

3. `-perm /mask`. Test bardzo podobny do poprzedniego, ale działający na zasadzie operatora `OR`, a nie `AND`. Tzn. jeśli w masce występuje kilka jedynek, to wystarczy, że co najmniej jedno z praw pliku odpowiadające tym jedynkom jest przyznane.

Z powyższego wynika, że jeśli w masce jest tylko jedna jedynka, to opcje `-perm -mask` oraz `-perm /mask` działają identycznie.

Wróćmy więc do naszego problemu tj. znalezienia plików, które przyznają prawo odczytu właścicielowi, ale nie przyznają prawa zapisu właścicielowi grupowemu. Sprawdzenie prawa odczytu dla właściciela jest dosyć proste:

```
-perm -400
```

taki zapis odpowiada prawom postaci `1** *** ***`. Druga część pytania jest trudniejsza, gdyż opcje `-perm -mask` i `-perm /mask` sprawdzają jedynie obecność jedynek, a nie obecność zer! Aby to obejść należy potraktować zera jako jedynki, a potem odwrócić wynik używając operatora `NOT` tzn.:

```
-perm -020
```

odpowiada prawom `*** *1* ***`, a po zanegowaniu:

```
 ! -perm -020
```

odpowiada prawom `*** *0* ***`, zgodnie z oczekiwaniem. Teraz możemy połączyć obie uzyskane opcje `-perm` z użyciem operatora `AND`:

```
-perm -400 -and ! -perm -020
```

co daje oczekiwane prawa postaci `1** *0* ***`.