Linuksowy Poradnik Olimpiady Informatycznej

Marek Żylak

21 marca 2006

1 Wstęp

Dokument ten powstał m.in. na podstawie doświadczeń studentów zdobytych podczas zawodów programistycznych o charakterze algorytmicznym (ACM, MPPZ, MWPZ itp.). Opisuje on podstawy obsługi Linuksa oraz sposoby na wydajne rozwiązywanie zadań olimpijskich przy pomocy standardowych narzędzi.

2 Czytanie i analiza zadania

Przeczytaj uważnie zadanie. Zaprojektuj algorytm i struktury danych. Opcjonalnie sporządź notatki na papierze. Najlepiej nie zabierać się za kodowanie, dopóki nie ma się w głowie pełnego obrazu rozwiązania. Podczas kodowania trzeba się koncentrować na poprawności programu, a nie wymyślaniu algorytmu.

3 Kodowanie

Zaloguj się na konsolę tekstową (przejście z trybu graficznego do tekstowego: **Ctrl-Alt-F1**) lub otwórz terminal graficzny (np. `xterm`a lub Konsolę).

Na komputerach udostępnianych zawodnikom Olimpiady Informatycznej, po włączeniu zasilania, domyślnie uruchamia się Linuks w trybie graficznym oraz menadżer okien KDE. W trybie graficznym zarządzanym przez KDE programy okienkowe można wybierać z menu rozwijanego przyciskiem „K" umiejscowionym w lewym dolnym rogu ekranu, który pełni rolę podobną jak przycisk „Start" w Windows. Aby uruchomić `kconsole`, rozwijamy główne menu klikając w przycisk „K", następnie rozwijamy podmenu „System" i w końcu wybieramy „Program Terminala (Konsola)".

Obok przycisku „K" znajdują się także inne przydatne przyciski. Jednym z nich jest przycisk „Pliki osobiste", który służy do przeglądania katalogu domowego należącego do aktualnie zalogowanego użytkownika.

Linuks jest systemem operacyjnym, w którym wielu użytkowników może pracować jednocześnie, dlatego każdy użytkownik posiada swój katalog domowy, który jest miejscem przechowywania wszystkich plików związanych z danym użytkownikiem (dokumenty, pliki konfiguracyjne, e-maile itp.).

Jeśli do kodowania wybrałeś tryb graficzny, możesz dodatkowo dostosować do swoich potrzeb ustawienia terminala (wielkość czcionki, układ kolorów). W trybie tekstowym masz dostępnych kilka wirtualnych konsol, między którymi można przełączać się przy pomocy kombinacji **Alt-F1**, **Alt-F2**, itd. Na jednej konsoli można mieć otwarty program, a na innej przykładowo dokumentację jakiejś funkcji (np. `man qsort`).

Po otworzeniu terminala aktualny katalog roboczy będzie ustawiony na twój katalog domowy. Do katalogu domowego można szybko wrócić przy pomocy polecenia:

```
$ cd
```

Sprawdzanie aktualnego katalogu roboczego odbywa się przy pomocy polecenia `pwd`:

```
$ pwd
/home/contest
```

Aby wypisać zawartość katalogu trzeba użyć:

```
$ ls
```

Jeśli chcemy dodatkowo wypisać rozmiary plików sformatowane w postaci bardziej czytelnej dla człowieka, należy użyć polecenia `ls` z opcjami `-l` oraz `-h`:

```
$ ls -lh
```

Szczegółowe opisy wszystkich poleceń Linuksowych można uzyskać przy pomocy polecenia `man` w następujący sposób:

```
$ man ls
```

Po wywołaniu opisu polecenia, nawigacja odbywa się przy pomocy strzałek oraz przycisków **PgUp** i **PgDown**. Wychodzi się przez naciśnięcie **q**. Wyszukiwanie tekstu jest możliwe po naciśnięciu / (działa też w wielu innych programach m.in. w Firefox). Jeśli po znalezieniu wzorca chcemy znaleźć kolejne jego wystąpienie trzeba, nacisnąć **n**. Czasami bardziej szczegółowe informacje można uzyskać dzięki poleceniu `info`:

```
$ info ls
```

Zalecane jest zapoznanie się z możliwościami podstawowych poleceń (`info coreutils`).

Utwórz podkatalog w którym będziesz trzymał wszystkie pliki (źródła, testy) dotyczące aktualnie rozwiązywanego zadania i przejdź do nowo utworzonego katalogu:

```
$ mkdir zad1
$ cd zad1
```

Bardzo pożyteczna funkcja linii poleceń jest automatyczne dopełnianie. Jeśli wpiszemy polecenie, którego argumentem jest nazwa pliku lub katalogu, to często argumentu nie trzeba wpisywać do końca. Wystarczy wpisać jego pierwsze litery, a potem nacisnąć **Tab**. Jeśli jest tylko jeden plik lub katalog, którego nazwa zaczyna się od podanych znaków, argument zostanie od razu dopełniony do tej właśnie nazwy! Jeśli jest kilka możliwości, zostaną one wypisane na ekranie i trzeba wtedy podać kolejne znaki argumentu. Więcej informacji o interpreterze linii poleceń można uzyskać po wywołaniu `man bash`.

Rozpocznij edycję pliku z rozwiązaniem. Do edycji najlepiej używać edytora z kolorowaniem składni. Najczęściej używane edytory to:

- **mcedit** — intuicyjny, obsługa jak w edytorach używanych pod Windows, można używać bez wiedzy wstępnej, ponieważ zawsze wyświetla na ekranie informacje na temat swoich funkcji i skrótów klawiszowych, zupełnie wystarczający na Olimpiadzie Informatycznej
- **vim** — nieintuicyjny interfejs, obsługi trzeba uczyć się z instrukcji i ćwiczyć kilka dni zanim nabierze się wprawy, umożliwia wykonanie skomplikowanych operacji przy pomocy kilku przyciśnięć klawiatury, bardzo dużo opcji, wbudowany język skryptowy, jego zalety docenić można raczej dopiero przy pisaniu dużej ilości programów i administracji systemu, niż na Olimpiadzie Informatycznej
- **emacs** — jeszcze bardziej rozbudowany niż vim, ale trochę bardziej intuicyjny, też wymaga nauki z podręcznika, wspomaga debugowanie
- **kwrite**, **kate** — standardowe edytory w środowisku graficznym KDE, intuicyjne oraz zawierające dużo opcji dostępnych przez menu, polecane w trybie graficznym jako alternatywa dla mcedit

Zaczynamy edycję nowego pliku przy pomocy polecenia:

```
$ mcedit zad1.cpp
```

4 Kompilacja

Po zapisaniu programu i wyjściu z edytora przystępujemy do kompilacji:

```
$ c++ zad1.cpp
```

Jeśli kompilacja nie powiodła się, analizujemy wypisane przez kompilator informacje o błędach i wracamy do edytora aby poprawić program. W przeciwnym wypadku powstanie plik wykonywalny `a.out`.

Można skompilować program w bardziej wyrafinowany sposób:

```
$ c++ -Wall -Wshadow -o zad1 zad1.cpp
```

Opcja `-Wall` włącza dodatkowe ostrzeżenia pomagające wykrywać błędy na etapie kompilacji, natomiast opcja `-Wshadow` nakazuje kompilatorowi wykrywanie przypadków przykrycia w programie jednych zmiennych przez inne zmienne o tej samej nazwie. Opcja `-o` ustawia nazwę pliku wykonywalnego.

Jeśli chcemy skompilować program z dodatkowymi optymalizacjami kodu maszynowego, można użyć polecenia:

```
$ c++ -Wall -Wshadow -O2 \
-fomit-frame-pointer -o zad1 zad1.cpp
```

Jeśli chcemy podzielić długie polecenie na kilka linii to umieszczamy znak \ na końcu każdej linii, oprócz ostatniej.

W tym miejscu warto powiedzieć, że kolejnym udogodnieniem Linuksowej linii poleceń jest możliwość szybkiego przywołania wcześniej wykonanych poleceń przy pomocy wciskania na

klawiaturze strzałki w górę. Przywołane polecenie można przed użyciem dodatkowo zmodyfikować.

5 Testowanie

Skompilowany plik wykonywalny `zad1` uruchamia się w następujący sposób:

```
$ ./zad1
```

Jeśli nasz program próbuje coś czytać ze standardowego wejścia, zatrzyma się w oczekiwaniu na dane zaraz po uruchomieniu. Można wtedy po prostu ręcznie wpisać odpowiednie dane. Koniec pliku uzyskuje się przez naciśnięcie **Ctrl-D**.

Gdy chcemy bezzwłocznie zakończyć aktualnie wykonywany program, używamy kombinacji **Ctrl-C**. Jeśli proces wyłączył możliwość zakończenia w taki sposób, trzeba uzyskać dostęp do innego terminala, znaleźć numer procesu przy pomocy polecenia `ps` lub `ps aux` (druga wersja daje więcej informacji) i zakończyć proces poleceniem:

```
$ kill -9 numer_procesu
```

Najczęściej można zakończyć proces dużo prościej:

```
$ killall -9 mc
```

To zakończy wszystkie procesu użytkownika, których pliki binarne miały nazwę mc (mc - Midnight Commander to bardzo intuicyjny, najczęściej używany tekstowy menadżer plików w Linuksie). Natomiast polecenie `killall mc` jest równoważne z próbą zakończenia wszystkich procesów użytkownika o nazwie `mc` przy pomocy kombinacji **Ctrl-C**.

Jeśli dane dla naszego programu zajmują tylko jedną linijkę, można je podać także w taki sposób:

```
$ echo "1 2 3 4" | ./zad1
```

Użyłem tu jednej z najprzydatniejszych funkcji linii poleceń, czyli potoków. Jeśli napiszemy kilka poleceń oddzielonych znakami `|` to standardowe wyjście pierwszego polecenia zostanie połączone ze standardowym wejściem drugiego polecenia itd. Czyli jeśli chcemy np. policzyć ile jest w aktualnym katalogu plików z literą `x` w nazwie, piszemy:

```
$ ls | grep x | wc -l
```

Taki sam efekt jaki daje ostatnie polecenie, można uzyskać przy pomocy mechanizmu wzorców:

```
$ ls *x* | wc -l
```

Jeśli w argumencie użyjemy znaku `*`, to jest on rozwijany do listy plików w aktualnym katalogu, których nazwę da się uzyskać przez zastąpienie każdej `*` we wzorcu przez pewien ciąg znaków.

Można także uruchomić program podając mu na wejściu zawartość jakiegoś pliku:

```
$ cat in | ./zad1
```

Jeszcze krócej można to zrobić w taki sposób:

```
$ ./zad1 < in
```

Taki zapis powoduje, że nie są uruchamiane żadne dodatkowe programy i standardowe wejście naszego programu jest połączone bezpośrednio z danym plikiem.

Wiemy już dużo o sterowaniu wejściem, ale nie zajmowaliśmy się do tej pory wyjściem programu. Normalnie program wypisuje wszystkie dane na ekran. Jeśli wypisał więcej linii niż posiada nasz terminal, dane, które zdążyły „uciec” przez górną krawędź terminala, można dzięki kombinacji **Shift-PgUp**. Możemy też zapisać dane wypisywane przez program do pliku (w naszym przypadku do `out`), aby móc się im dokładniej przyjrzeć w edytorze i ewentualnie dalej przetwarzać:

```
$ ./zad1 < in > out
```

Jeśli nie chcemy kasować zawartości pliku `out` tylko dopisać dane na końcu, musimy użyć innego operatora przekierowania wyjścia:

```
$ ./zad1 < in >> out
```

Można też zapisać dodatkowo do pliku standardowe wyjście diagnostyczne programu (stderr):

```
$ ./zad1 < in > out 2> err
```

Jeśli chcemy mierzyć wydajność programu, na pewno ważna jest możliwość mierzenia czasu jego wykonania:

```
$ time ./zad1 < in > out
```

Po wykonaniu małych testów poprawnościowych warto też przetestować program na większych danych. Przydają się wtedy różne generatory testów.

Podczas testowania, szczególnie kiedy powstają duże pliki wynikowe, czasami pojawia się konieczność porównania dwóch plików. Można to zrobić przy pomocy polecenia `cmp`:

```
$ cmp plik1 plik2
```

Jeśli chcemy dostać dokładniejsze dane o tym, w których miejscach pliki różnią się, lepiej jest użyć:

```
$ diff plik1 plik2
```

Jeśli podczas testowania okazało się, że program nie działa zgodnie z naszymi oczekiwaniami, trzeba jak najszybciej znaleźć i poprawić błąd. Są dwie szkoły debugowania.

Pierwsza metoda nie potrzebuje debugera i polega modyfikowaniu programu w celu znalezienia błędów, a dokładniej, dodawaniu instrukcji które sprawdzają poprawność obliczeń i spójność struktur danych oraz wypisują dodatkowe informacje o tym co dzieje się w trakcie obliczeń.

Można np. dodać instrukcje, które będą wypisywać stan wybranych zmiennych, w wybranych momentach wykonania programu:

```
int a[3][3],x;
(...)
printf("x=%d\n",x);
for (int i=0;i<3;i++) {
  for (int j=0;j<3;j++)
    printf("%3d",a[i][j]);
  printf("\n");
}
```

Można też dodać kod, który będzie testował, czy zmienne i struktury danych spełniają określone warunki i wypisywał komunikaty jeśli coś idzie nie tak:

```
if (x<0) {
  printf("błąd: x=%d\n",x);
  exit(1);
}
```

Można też to zrobić z użyciem biblioteki `assert.h`:

```
#include <assert.h>
(...)
assert(x>=0);
```

W tym przypadku, jeśli warunek nie zostanie spełniony, program zatrzyma się i wypisze numer linijki w której asercja nie została spełniona.

Druga metoda debugowania wymaga użycia debugera. Standardowym debugerem pod Linuksem jest `gdb`. Jeśli zamierzamy używać `gdb`, kod binarny naszego programu musi zawierać dodatkowe informacje. Odpowiedni kod binarny uzyskuje się przez przekompilowanie programu z opcją `-g`:

```
$ c++ -g -o zad1 zad1.cpp
```

Przy pomocy `gdb` można też debugować odpowiednio skompilowane programy w Pascalu:

```
$ fpc -g zad1.pas
```

Uruchamiamy `gdb` i ładujemy nasz plik binarny:

```
$ gdb zad1
```

Po uruchomieniu `gdb` znajdziemy się w linii poleceń debugera. Możemy od razu przeglądać kod programu przy pomocy polecenia `list` lub w skrócie `l`. Jeśli chcemy zobaczyć fragment programu otaczający wybraną linię wystarczy wpisać:

```
(gdb) l N
```

gdzie `N` to numer linii programu. Możemy także wyświetlić treść dowolnej funkcji:

```
(gdb) l nazwa_funkcji
```

Aby wystartować nasz program, trzeba użyć polecenia `run` (lub `r`). Bez dodatkowych ustawień program wykona się do końca zatrzymując się tylko w oczekiwaniu na dane. Przy uruchamianiu można od razu przekierować zawartość dowolnego pliku na standardowe wejście naszego programu:

```
(gdb) r < in
```

Jeśli chcemy mieć możliwość wykonywania programu linijka po linijce, konieczne jest ustawienie tzw. punktów zatrzymania (*breakpoint*). Aby ustawić punkt zatrzymania trzeba użyć polecenia `break` lub w skrócie `b`:

```
(gdb) b N
```

Można ustawić kilka punktów zatrzymania, nie tylko związanych z konkretnych liniami programu, ale także z wywołaniami wybranych funkcji:

```
(gdb) b moja_funkcja
```

Jeśli chcemy zatrzymać nasz program od razu po uruchomieniu, piszemy:

```
(gdb) b main
```

To samo dotyczy programów w Pascalu.

Po ustawieniu punktów zatrzymania, uruchamiamy nasz program i czekamy na jego zatrzymanie. Jeśli zatrzymał się na punkcie zatrzymania, na ekranie zostanie wyświetlona linia, która ma być za chwilę wykonana. Aby ją wykonać i przejść do kolejnej linii, używamy polecenia `next` (albo `n`). Aby powtórzyć ostatnio wykonane polecenie nie musimy go wpisywać. Wystarczy wcisnąć **Enter**. Debuger powtórzy wtedy ostatnio użyte polecenie.

W celu kontynuacji do kolejnego punktu zatrzymania, używamy polecenia `continue` (lub `c`).

Gdy w linii, która ma się wykonać jest wywołanie funkcji i chcemy prześledzić dokładnie to wywołanie, trzeba zamiast `next` użyć polecenia `step` (lub `s`).

W trybie pracy krok po kroku można oglądać aktualne wartości zmiennych, w tym tablic i napisów. Robi się to przy pomocy polecenia `print` (lub p):

```
(gdb) p x
```

Dodatkowo można ustawiać punkty obserwacyjne (*watchpoint*), przy pomocy polecenia `watch` (lub `w`). Powodują one wypisanie stosownej informacji przy każdej zmianie obserwowanej zmiennej. Da się nawet zmieniać wartości zmiennych i wywoływać funkcje (polecenia `set` i `call`)!

Więcej informacji o możliwościach `gdb` można uzyskać po wejściu do tego programu i wpisaniu polecenia `help`.

6 Automatyzacja pracy

W pierwszej kolejności warto usprawnić kompilację. Podstawowa metoda polega na utworzeniu odpowiedniego pliku `Makefile`:

```
all: zad1
zad1: zad1.cpp
    c++ -g -Wall -Wshadow \
      -o zad1 zad1.cpp
```

Stworzony plik jest używany przez program `make`, który służy do automatyzacji procesu kompilacji. Wystarczy teraz wpisać:

```
$ make
```

Program `make` sprawdzi, czy kod binarny jest aktualny w stosunku do kodu źródłowego. Jeśli nie jest aktualny, przekompiluje program.

W `Makefile` można dodać też funkcje odpowiedzialne za testowanie:

```
t: zad1
time ./zad1 < in | tee out
```

Teraz możemy wywołać polecenie:

```
$ make t
```

Wywołanie takiego polecenia uaktualnia w razie potrzeby kod binarny naszego programu, a potem wykonuje go podając na wejściu zawartość pliku `in` i zapisując wyjście programu do pliku `out` oraz na ekran.

Testowanie można także zautomatyzować wykorzystując elementy programowania interpreterze linii poleceń `bash`. Po napisaniu poniższego polecenia zostaną znalezione wszystkie pliki w aktualnym katalogu roboczym, których nazwy kończą się znakami `.in`. Następnie nasz program zostanie wykonany kilka razy, za każdym razem dostając na wejściu kolejny plik. Wyniki wykonań programu znajdą się w plikach o nazwach takich jakie miały pliki wejściowe, tylko z rozszerzeniami zmienionymi z `.in` na `.out`:

```
$ for f in *.in; do \
./zad1 < $f > `basename $f .in`.out \
|| echo "Błąd: $f"; done
```

Większe polecenia warto zapisać sobie w plikach (skryptach):

```
#!/bin/bash
for f in *.in
do
  ./zad1 < $f > `basename $f .in`.out \
    || echo "Błąd: $f"
done
```

Po utworzeniu skryptu trzeba mu jeszcze nadać uprawnienie do wykonania:

```
$ chmod +x testy
```

Tak przygotowany skrypt jest gotowy do użycia:

```
$ ./testy
```

W skryptach można też używać pętli w stylu języka C:

```
#!/bin/bash
for ((i=0;i<10;i++))
do
  echo $i | ./zad1
done
```

Linuksowej linii poleceń można także używać jako prostego kalkulatora:

```
$ echo $((2+(3*7)))
23
```

W większości dystrybucji Linuksa standardowo jest dostępny kalkulator o sporych możliwościach. Chodzi mianowicie o program `bc`. Pozwala on na obliczenia na liczbach całkowitych o dziesiątkach tysięcy cyfr oraz konwersję między systemami liczbowymi o różnych podstawach. Oto przykładowe użycie `bc`:

```
$ echo "2^100" | bc
1267650600228229401496703205376
```