Program make

Opracował: dr inż. Zbigniew Jaworski

1. Wstęp

Program **make** jest standardowym narzędziem dostępnym w środowisku systemu UNIX ułatwiającym programiście pracę nad tworzeniem programu. Podstawową funkcją programu **make** jest zarządzanie kompilacją zbioru tekstów źródłowych składających się na dany program przy zachowaniu minimalnego kosztu operacji. Oznacza to, że po zmodyfikowaniu części tekstów źródłowych, kompilacji zostaną poddane wyłącznie te moduły, które są od nich uzależnione. O tym, jakie polecenia należy wykonać, aby z tekstów źródłowych otrzymać programy wykonywalne decydują reguły transformacji. Reguły te są wstępnie zdefinione. Użytkownik może też tworzyć własne reguły, które uzupełnią lub zmienią zestaw standardowy. Reguły wraz z opisem sposobu uzyskania programu wynikowego umieszcza się w pliku sterującym. Plik sterujący powinien znajdować się w katalogu, który zawiera teksty źródłowe danego programu.

Uruchomienie procesu kompilacji następuje poprzez wywołanie programu **make**. Program odczytuje wtedy całą zawartość pliku sterującego i na podstawie jego treści, jak również na podstawie wbudowanych reguł transformacji, treści linii wywołania, wartości zmiennych środowiska, czasu systemowego oraz czasu modyfikacji plików tworzy wynikowy ciąg poleceń prowadzący do uzyskania żądanego celu. Następnie polecenia te są wykonane, przy czym wykonanie każdego z nich poporzedzone jest wypisaniem jego treści.

2. Parametry wywołania

Wywołanie programu **make** ma następującą postać:

```
make [opcje] [makrodefinicje] [-f plik_sterujący] [cel]
```

W najprostszym przypadku linia polecenia zawiera tylko słowo **make**. Program próbuje wtedy odczytać polecenia z pliku sterującego o standardowej nazwie: **`makefile`**, **`Makefile`** lub **`MakeFile`**. Jeśli w bieżącym katalogu nie ma pliku o takiej nazwie to zgłaszany jest błąd. Jeśli plik sterujący nosi inną nazwę to należy użyć wywołania

```
make -f nazwa_pliku_sterującego
```

W linii wywołania można zdefiniować nowe lub zmienić istniejące już makrodefinicje, np.:

```
make "CC=gcc"
```

Można też polecić osiągnięcie innego celu niż domyślny, np.:

make all, make clean lub **make install.**

2.1. Najczęściej używane opcje

Spośród wielu opcji (o znaczeniu których można się dowiedzieć wydając polecenie: **man make**) do najczęściej używanych należą :

-d	włącza tryb szczegółowego śledzenia,
-f plik_sterujący	umożliwia stosowanie innych niż standardowe nazw plików sterujących,
-i	powoduje ignorowanie błędów kompilacji (stosować z ostrożnością!),
-n	powoduje wypisanie poleceń na ekran zamiast ich wykonania,
-p	powoduje wypisanie makrodefinicji i reguł transformacji,
-s	wyłącza wypisywanie treści polecenia przed jego wykonaniem.

Opcje można ze sobą łączyć. Np.: polecenie **make -np** powoduje wypisanie wszystkich reguł i makrodefinicji oraz ciągu poleceń jakie powinne być wykonane, aby uzyskać żądany cel. Jest to pomocne w sytuacji, gdy programista chce sprawdzić poprawność definicji zawartych w pliku sterującym bez uruchamiania długotrwałej kompilacji wielu plików.

3. Plik sterujący

Plik sterujący zawiera definicje relacji zależności, które mówią w jaki sposób i z jakich elementów należy stworzyć cel (program, bibliotekę, lub plik obiektowy) i wskazują pliki, których zmiany implikują wykonanie powtórnej kompilacji poszczególnych celów. Plik sterujący może również zawierać zdefiniowane przez programistę reguły transformacji.

3.1. Definicja relacji zależności

Program **make**, stosując wbudowane reguły transformacji, potrafi samodzielnie wykonać proste sekwencje poleceń, ale potrzebuje wskazówek programisty by utworzyć bardziej skomplikowane cele, takie jak program wykonywalny. Programista dokonuje tego

poprzez umieszczenie w pliku sterującym definicji określających, z jakich elementów należy tworzyć program wynikowy i jak te elementy zależą od innych obiektów np. plików nagłówkowych. Wszystkie relacje zależności pomiędzy obiektami opierają się na porównywaniu czasu ostatniej modyfikacji plików oraz na sprawdzaniu czy dane pliki istnieją. Ogólna postać definicji, jaką można umieścić w pliku sterującym, jest następująca:

```
cel1  [cel2...] :[:]  [obiektyodniesienia]
[<TAB> polecenia]  [#komentarz]
        ⋮
[<TAB> polecenia]  [#komentarz]
```

(gdzie <TAB> oznacza znak tabulacji).
Każde polecenie umieszczone w definicji wykonywane jest w oddzielnej kopii shell'a. Standardowo jest to Bourne shell (sh). Jeśli użytkownik chce wywoływać inny rodzaj shell'a, to musi przedefiniować zmienną SHELL.

Złożone polecenia powinny być zawarte w pojedynczym wierszu, gdyż tylko wtedy wykonane będą przez tą samą kopię shell'a. Elementarne składniki polecenia należy umieścić obok siebie, oddzielając je średnikami, a jeśli cały wiersz jest zbyt długi, to należy rozbić go na kilka stosując znak kontynuacji - \:

```
prog : source/main.o source/input.o source/output.o
  cd source; $(CC)  -o prog  main.o input.o output.o

prog : source/main.o source/input.o source/output.o
  cd source; \
  $(CC)  -o prog  main.o input.o output.o
```

Jeśli użytkownik chce zapobiec wypisywaniu treści polecenia podczas jego wykonywania to może umieścić @ jako pierwszy znak w poleceniu.

Poniższy przykład pokazuje typową definicję relacji zależności:

```
  prog : main.o input.o output.o
       $(CC)  -o prog  main.o input.o output.o
```

Mówi ona, że cel **prog** zależy od trzech obiektów odniesienia: **main.o**, **input.o**, **output.o**. Jeśli którykolwiek z tych obiektów ulegnie zmianie (tzn. zostanie na nowo skompilowany), to cel **prog** należy utworzyć ponownie. Jeżeli obiekt odniesienia nie istnieje, to zostanie stworzony przy zastosowaniu wbudowanej reguły transformacji (lub innej definicji podanej przez programistę w pliku sterującym).W drugim wierszu definicji umieszczone jest polecenie, które należy wykonać by zbudować cel **prog**. W tym przypadku,

należy wywołać kompilator z opcją zmieniającą nazwę pliku wynikowego na nazwę celu i wykonać konsolidację plików **main.o**, **input.o**, **output.o** z biblioteką standardową.

Programista nie musi umieszczać w pliku sterującym instrukcji powodujących ponowne tworzenie plików obiektowych w przypadku zmiany tekstów źródłowych, gdyż jest to wykonywane automatycznie. Musi natomiast poinformować program **make** o wszystkich plikach włączanych do tekstów źródłowych dyrektywą `#include`, aby **make** mógł zareagować na zmiany zawartości tych plików (wyjątkiem od tej zasady są standardowe pliki nagłówkowe kompilatora, które nigdy nie ulegają zmianom). Informacja ta podawana jest w następującej postaci:

```
main.o : global.h
main.o input.o : czytaj.h
output.o main.o : global.h data.h matrix.h
```

Definicje te mówią, że w przypadku zmiany jakiegokolwiek pliku (plików) po prawej stronie '**:**' należy od nowa stworzyć plik (pliki) wymienione po lewej stronie '**:**', stosując wbudowane reguły transformacji.

Program **make** inaczej interpretuje definicje zawierające separatory '**:**' oraz '**::**' .
W przypadku definicji zawierającej '**:**' cel jest budowany jeśli:
1) jakikolwiek obiekt odniesienia jest "młodszy" od celu,
2) cel nie istnieje.

Poniższy przykład, pokazuje użycie separatora '**::**'. Dwie definicje celu **a** umieszczone w jednym pliku sterującym możliwiają wykonania kompilacji "warunkowej":

```
a :: a.sh
     cp  a.sh  a

a :: a.c
     cc  -o a  a.c
```

Definicja pierwsza aktywna jest, gdy w bieżącym katalogu znajduje się plik **a.sh**, a druga jeśli plik **a.c**. W przypadku, gdy istnieją oba pliki, wykonana może być zarówno jedna z definicji lub też obie zależnie od tego czy cel jest "starszy" od **a.sh** czy od **a.c** czy jednocześnie od **a.sh** i **a.c**.
W następnym przykładzie użycia definicji z '**::**' cel ma nazwę pokrywającą się z nazwą katalogu zawierającego jego teksty źródłowe (sytuacja często spotykana w praktyce). Gdyby użyto definicji z pojedynczym '**:**' to cel nigdy nie zostałby utworzony, gdyż istnieje już w bieżącym katalogu obiekt o tej samej nazwie (to, że jest katalogiem nie ma znaczenia dla programu **make**).

```
program ::
    cd program; cc program.o -o program
```

Kolejność umieszczenia definicji w pliku sterującym nie wpływa na kolejność wykonywania poleceń przez program **make**. To, jakie czynności i w jakiej kolejność należy wykonać określa sam program **make** na podstawie wewnętrznej struktury danych (utworzonej w wyniku analizy całej treści pliku sterującego) i czasu ostatniej modyfikacji plików.
Wyjątkiem od tej zasady jest określenie, który z celów zdefiniowanych w pliku sterującym będzie realizowany domyślnie, czyli po uruchomieniu programu **make** bez podania nazwy celu. **Standardowo za domyślny przyjmowany jest pierwszy zdefiniowany cel.**

3.2. Makrodefinicje

Makrodefinicja jest to zmienna używana w celu sparametryzowania reguł, dzięki czemu stają się one bardziej przejrzyste i łatwiejsze do modyfikacji. Deklaracja makrodefinicji to linia zawierająca znak równości i nie zaczynająca się od kropki ani tabulatora, np.:

```
OBJECTS = main.o data.o input.o output.o
HDRS = ../headers ../includes
CFLAGS = -g -DAUX -I$(HDRS)
CC = gcc
LIBS = -lusux
```

W celu odwołania się do zawartości makrodefinicji używa się konstrukcji **$(nazwa_makrodef)** lub **${nazwa_makrodef}**. Jeśli nazwa makrodefinicji zawiera tylko jeden znak to można pominąć nawiasy. Stosowanie makrodefinicji daje możliwość łatwego wprowadzania modyfikacji takich jak zmiana opcji kompilatora, a także przystosowania pliku sterującego do nowego środowiska, np.: zmiana ścieżek dostępu do plików nagłówkowych.

Istnieje zestaw predefiniowanych makrodefinicji, do których programista może się odwoływać w pliku sterującym. Wartości tych makrodefinicji mogą być zmienione, przy czym nowe wartości widoczne są tylko w konkretnym pliku sterującym. Oto niektóre z nich:

Makrodefinicja	Wartość predefiniowana	Znaczenie
AR	**ar**	program zarządzający bibliotekami
AS	**as**	asembler

ASFLAGS		opcje programu AS
CC	**cc**	kompilator języka C
CFLAGS		opcje programu CC
LD	**ld**	konsolidator
LDFLAGS		opcje programu LD

Istnieje również zbiór wbudowanych makrodefinicji, których wartości są określane w trakcie wykonywania poleceń zapisanych w definicjach (tzw. makrodefinicje dynamiczne). Stosowanie tych makrodefinicji znacznie ułatwia tworzenie definicji i reguł transformacji.

W definicjach można stosować następujące makra :

$@** - aktualnie tworzony cel; w poniższym przykładzie **$@ przyjmuje wartość **prog**:

```
prog : main.o input.o output.o
            $(CC)   -o $@ main.o input.o output.o
```

$?** - lista nieaktualnych obiektów odniesienia w stosunku do bieżącego celu; w poniższym przykładzie **$? jest listą tych plików obiektowych spośród wszystkich z **$(LIB_OBJ)**, które zostały zmodyfikowane po ostatnim utworzeniu biblioteki **libusux.a**:

```
libusux.a : $(LIB_OBJ)
      $(AR) rv $@ $?
```

$< - nieaktualny obiekt odniesienia powodujący wywołanie reguły, np.:

```
.c.o:
      $(CC) $(CFLAGS) -c $(<)
```

$* - wspólny prefix celu i obiektu odniesienia, np.:

```
.c.o:
      $(CC) $(CFLAGS) -c $*.c
```

Do powyższych makrodefinicji można zastosować modyfikatory **D** (ang. directory) oraz **F** (ang. file), umożliwiające odzyskanie z nazwy celu części opisującej katalog i części będącej właściwą nazwą, np:

$(@D) - część "katalogowa" nazwy bieżącego celu

$(@F) - część "plikowa" nazwy bieżącego celu

Istnieją również makrodefinicje, które umieszczać wolno wyłącznie na liście obiektów odniesienia (po '**:**' lub '**::**'). Są to :

$$@ - kolejny cel (obiekt z lewej strony znaku **:**), np.:

```
prog1 prog2 prog3 prog4 : $$@.c
    $(CC) -o $@ $?
```

Definicja ta powoduje wykonanie dla każdego z celów prog1, prog2, prog3, prog4 następującej sekwencji operacji :

1. ustawienie bieżącego celu na **prog*n***
2. ustawienie bieżącego obiektu odniesienia na **prog*n*.c**,
3. wywołanie kompilatora jeśli **prog*n*.c** jest "młodszy" niż **prog*n*** i utworzenie celu **prog*n***.

3.3. Hierarchia definiowania makrodefinicji

W chwili wywołania program **make** odczytuje zawartość zmiennych środowiska i dodaje je do zbioru makrodefinicji. Jeżeli nazwy makrodefinicji użytych wewnątrz pliku sterującego pokrywają się z nazwami użytymi w linii wywołania i/lub z nazwami zmiennych środowiska, to ostateczna wartość makrodefinicji wynika z następującej hierarchii (w kolejności od najwyższego do najniższego priorytetu):

1. wartość nadana w linii wywołania,
2. wartość zdefiniowana w pliku sterującym,
3. wartość zmiennej środowiska,
4. predefiniowana wartość makrodefinicji.

3.4. Reguły transformacji

Program **make** posiada wewnętrzną tablicę reguł transformacji, które są wykorzystywane podczas kompilacji. Użytkownik może dodatkowo zdefiniować w pliku sterującym własne reguły, które "przysłonią" reguły wbudowane.

Reguły określają w jaki sposób dokonać przejścia od plików zawierających teksty źródłowe do plików obiektowych i do programów wynikowych. Typy plików określane są

na podstawie przyrostków (rozszerzeń nazw). Zdefiniowany jest standardowy zestaw tych przyrostków i skojarzonych z nimi typów plików. Istotniejsze z nich to:

Przyrostek	Przypisane znaczenie	Przykład
.c	tekst źródłowy w C	**main.c**
.h	plik nagłówkowy	**stdio.h**
.i	tekst po prekompilacji	**main.i**
.s	kod w języku asemblera	**main.s**
.o	kod relokowalny (wynik asemblacji)	**parser.o**

Istnieją dwa typy reguł transformacji: **dwuprzyrostkowe** i **jednoprzyrostkowe**.
Reguły dwuprzyrostkowe określają sekwencję poleceń jakie trzeba wykonać, aby przejść od pliku o typie skojarzonym z pierwszym przyrostkiem do pliku o typie skojarzonym z drugim przyrostkiem. Typowym przykładem jest reguła definiująca w jaki sposób uzyskać plik obiektowy z pliku zawierającego kod źródłowy w C:

```
.c.o:
     $(CC) $(CFLAGS) -c $(<)
```

Reguła ta mówi, że należy wywołać kompilator języka C z opcją zatrzymującą kompilację po etapie asemblacji, podając jako argument plik z tekstem źródłowym w C.
Podobną postać mają reguły określające transformacje plików zawierających teksty źródłowe dla asemblera :

```
.s.o:
     $(AS) $(ASFLAGS) -o $(@) $(<)
```

Reguły jednoprzyrostkowe określają sekwencję poleceń jakie należy wykonać, aby przejść od pliku o typie skojarzonym z danym przyrostkiem do pliku bez przyrostka czyli np.: programu wynikowego. Typowy przykład to reguła definiująca sposób uzyskania programu, przyjmując za wejście plik z tekstem źródłowym w C :

```
.c:
     $(CC) $(CFLAGS) $(LDFLAGS) -o $(@) $(<)
```

3.5. Predefiniowane cele.

Predefiniowane cele są w rzeczywistości wbudowanymi regułami, które są uaktywniane poprzez włączenie ich do pliku sterującego. Włączenie ich modyfikuje standardowy sposób działania programu **make**. Cele te to:

.DEFAULT: umieszczenie powoduje, że w przypadku, gdy brak jest definicji opisującej sposób zbudowania celu (a cel musi być osiągnięty) to stosowane są polecenia umieszczone w definicji celu **.DEFAULT**,

.IGNORE: umieszczenie tego celu jest równoznaczne z wywołaniem programu **make** z opcją **-i**,

.MAKESTOP [n]: umieszczenie tego celu powoduje zignorowanie całej treści pliku sterującego. Opcjonalnie można podać kod zakończenia (argument **n**), który standardowo ma wartość 0. Użycie celu **.MAKESTOP** umożliwia zaniechanie wykonywania wybranego pliku sterującego w celu szybkiego przejścia przez wielopoziomową strukturę wywołań programu **make**.

.PRECIOUS: umieszczenie tego celu zmienia standardowe zachowanie programu **make** w sytuacji przerwania jego pracy, powoduje zaniechanie usuwania utworzonych do tego czasu programów i plików obiektowych.

.SILENT: umieszczenie tego celu jest równoznaczne z wywołaniem programu **make** z opcją **-s**.

3.6. Instrukcja include.

W pliku sterującym można umieścić instrukcję **include** (lub **Include**) o następującej składni:

```
include     nazwa_pliku
```

Słowo **include** musi być umieszczone na początku linii zaczynając od pierwszej kolumny. Plik wskazany przez instrukcję **include** jest włączany do treści bieżącego pliku sterującego, po etapie wykonania podstawień makrodefinicji w pliku bieżącym. Jeśli nie jest możliwe odczytanie zawartości wskazanego pliku to program **make** przerywa pracę.
Jeśli użyte było polecenie **include** to w przypadku wystąpienia błędu odczytu program **make** kontynuuje pracę.