Make

Poročilo seminarske naloge

Nikola Brković

15. januar 2025

Povzetek

Make je orodje za avtomatizacijo izgradnje programske opreme. To poročilo ponuja pregled nad osnovnimi funkcionalnostimi in koncepti delovanja, ter podrobnosti različice GNU Make.

Uvod

Izgradnja programske opreme je pri velikih projektih lahko zelo kompleksen in zamuden postopek. Postopek pretvorbe izvorne kode v neko končno obliko (npr. izvršljivo datoteko) najpogosteje poteka v več fazah. Z orodjem Make lahko postopek izgradnje opišemo v datoteki, ki se imenuje Makefile, kjer opišemo, kako določeno fazo izvesti in odvisnosti med posameznimi fazami.

Različice

Originalno različico programa Make je ustvaril Stuart Feldman v 70-ih letih [1], in je takrat delovala na zgodnjih različicah operacijskega sistema Unix. Danes pa poznamo več različic programa Make:

GNU Make Je danes najbolj razširjena različica Make, privzeta tako na Linuxu kot tudi MacOS-u. Predstavlja del operacijskega sistema GNU. Deluje pa tudi na Windowsu, zlasti se uporablja v unixaških okoljih na Windowsu (Cygwin/MSYS2).

BSD Make Je privzeta različica v operacijskih sistemih družine BSD (FreeBSD, NetBSD).

NMAKE (Microsoft Program Maintenance Utility) Je različica, katero je ustvaril Microsoft in teče na Windowsu. Je podprta v razvojnem okolju Visual Studio. [4]

dmake (Distributed Make) Je različica, katero razvija Apache Software Foundation, in se uporablja pri projektu OpenOffice [2]. Predhodno jo je razvijalo podjetje Sun.

Ker je to najbolj priljubljena različica, se bom v tem poročilu osredotočil na GNUjevsko različico, natančneje na izdajo 4.4.1. Kosi izvorne kode in podrobnosti implentacije bodo izvirali iz te različice, vendar splošni koncepti veljajo tudi za druge različice.

Osnovna sintaksa

Makefile (poleg drugih zadev, kot so definicije spremenljivk) v osnovi sestavljajo definicije ciljev. Definicija enega (ali več enakih ciljev) izgleda tako:

```
cilj [cilj ...]: [predpogoj ...]
 [prvo pravilo]
 ...
 [zadnje pravilo]
```

V večini primerov predstavlja ime cilja relativno pot (od Makefila) do datoteke (ali mape). Takim ciljem pravimo tudi ciljne datoteke (ang. *target file*). Ciljem, ki ne predstavljajo datoteke (ali mape), pa pravimo umetni cilji (ang. *phony targets*).

Predpogoji (oziroma odvisnosti) so drugi cilji, od katerih je definirani cilj odvisen, kar pomeni, da se morajo uspešno izvesti, preden se definirani cilj izvede.

Vsako pravilo predstavlja stavek, ki se lahko izvede v sistemski lupini. Ker GNU Make izvira iz unixaškega sveta, je razvit z unixaškimi lupini v mislih, ampak podpira tudi neunixaške lupine, kot so `cmd.exe` (Windows NT), `command.com` (DOS, Windows 95 in 98), `4OS2` (OS/2). V stavek lahko vključimo določene izraze, katere bo Make izračunal ob izvajanju, ampak je sintaksa samih pravil precej odvisna od operacijskega sistema in izbrane sistemske lupine.

Odvisnosti

V osnovi predstavlja Makefile tekstovno reprezentacijo acikličnega usmerjenega grafa odvisnosti. Vsak cilj predstavlja eno vozlišče v grafu, odvisnosti so pa usmerjene povezave.

Ko Make zaženemo, lahko podamo cilj, ki naj bi se izvedel, ali pa lahko nastavimo privzeti cilj v Makefilu. V nadaljevanju bomo temu cilju pravili začetni cilj. Ob vsakem zagonu, Make bo izvedel samo tiste cilje v grafu odvisnosti, ki jih je potrebno izvesti.

Da Make ugotovi, ali je nek cilj potrebno ponovno izvesti, se zanaša na čas spremembe datoteke (*mtime*). V GNU Make je razreševanje odvisnosti (*dependency engine*) implementirano v datoteki *src/remake.c*. Za enostavne Makefile deluje algoritem približno tako:

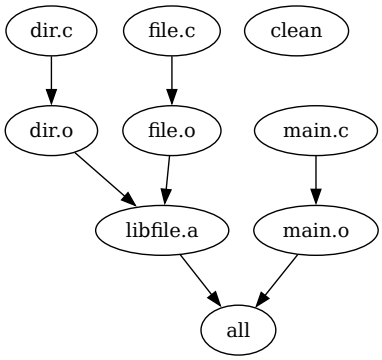
1. Če je začetni cilj umeten, in nima nobenih predpogojev, se bo izvedel in se postopek konča.

2. Spustimo se po grafu odvisnosti, od vključno trenutnega cilja, v nasprotni smeri povezav, dokler ne naletimo na cilj, katerega čas zadnje spremembe je starejši kot čas zadnje spremembe enega od njegovih predpogojev.

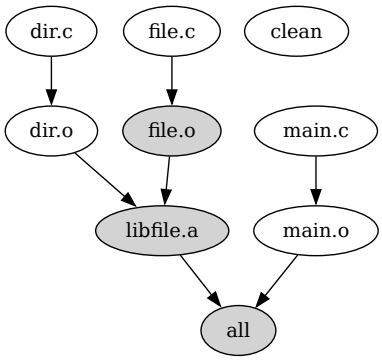
3. Če ni nobenega zastarelega cilja v celotnem grafu, se postopek konča, sicer pa najdeni cilj izvedemo.

4. Vrnemo se v prejšnji cilj, poskušamo ponoviti korak 2 čim večkrat. Preskočimo cilje, ki smo jih morebiti že izvedli. Če zastarelih ciljev ne najdemo več, izvedemo trenutni cilj in ponovimo ta korak.

5. Končamo, ko smo prišli nazaj v začetni cilj.



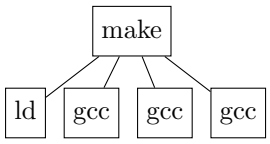
Slika 1: Primer grafa odvistnosti za preprosti Makefile. “all” pa “clean” sta umetna cilja, ostali so datoteke.



Slika 2: Če posodobimo datoteko file.c, in zaženemo "make all", bodo izvedeni cilji, označeni s sivo barvo.

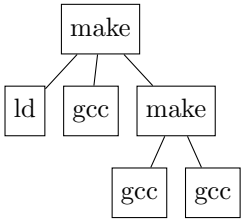
Vzporedno izvajanje poslov

Pri izvajanju programa Make, mu lahko podamo dodatno zastavico `-j` (npr. `make -j 32`), ki nastavi največje število poslov, ki jih Make lahko izvaja hkrati. Temu število pravimo tudi število poslovnih rež (ang. *job slots*).



Slika 3: Primer izvajanja Make s štirimi poslovnimi režami

Težave pri vzporednem izvajanju poslov povzroča rekurzivna uporaba Make (ang. *recursive make*). Rekurzivna uporaba pomeni, da se znotraj nekega pravila v Makefilu ponovno kliče Make. Čeprav je takšna uporaba odsvetovana ("Recursive Make Considered Harmful") [5], se še vedno uporablja pri večjih projektih, ki so sestavljeni iz več datotek z izvorno kodo.



Slika 4: Primer izvajanja rekurzivnega Make s štirimi poslovnimi režami

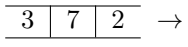
```
subsystem:
    cd subdir && $(MAKE)
```

Slika 5: Primer rekurzivnega izvajanja Make

V primeru, ko na enem sistemu hkrati teče več procesov Make, je težko spremljati, koliko poslov se dejansko izvaja v vseh procesih skupaj. Pri GNU Make so ta problem rešili s komponento, ki se imenuje *jobserver* in temelji na medprocesni komunikaciji (IPC) [6].

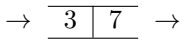
Implementacija jobserverja se razlikuje glede na operacijski sistem. Na sistemih, ki ustrezajo standardu POSIX, je jobserver implementiran z uporabo poimenovane cevi (ang. *named pipe*, v unixaški terminologiji tudi *FIFO*). Na sistemih POSIX, ki ne podpirajo poimenovanih cevi, mora starš podati otroškim procesom opisnik datoteke. Deluje pa na preprostem principu:

1. V primeru, ko imamo N poslovnih rež, bo korenska instanca Make (tista, ki ni klicana rekurzivno), v cev zapisala $N - 1$ žetonov. Žeton je predstavljen z enim bajtom. Nima vnaprej določene vrednosti, vsi žetoni so lahko popolnoma enaki.



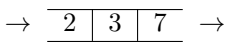
Slika 6: Začetno stanje cevi

2. Ko želimo zagnati novi proces, moramo "zasesti režo" oziroma prebrati en žeton iz cevi. Načeloma želimo, da bo to branje blokirajoče, kar pomeni da se iz sistemskega klica `read` ne bomo vrnili, dokler ne bo žeton na voljo. Implementirano v funkciji `jobserver_acquire` v izvorni datoteki `src/posixos.c`.



Slika 7: Stanje cevi po prebranem žetonu

3. Ko smo posel končali, moramo pa "sprostiti režo" oziroma zapisati žeton nazaj v cev. Pri tem pa mora vrednost zapisanega žetona biti enaka tisti, ki smo jo prebrali, ko smo posel ustvarili. Implementirano v funkciji `jobserver_release` v izvorni datoteki `src/posixos.c`.



Slika 8: Stanje cevi po zapisanem žetonu

Vendar, pri takšni implementaciji obstaja ena pomanjkljivost. Če smo v situaciji, kjer želimo zagnati novi posel, lahko čakamo, da se bo sprostila reža pri jobserverju. Med tem čakanjem se lahko zgodi, da se konča nek posel, katerega je ustvaril trenutni proces Makea. Če je čakanje na režo v sistemskem klicu `read` blokirajoče, potem se proces ne more ustrezno odzvati na signal `SIGCHLD`, ki sporoči, da se je otroški proces posla končal. [1]

Pri starejših sistemih POSIX, je Make to težavo respil tako, da, preden začne čakati na žeton, poleg že obstoječega opisnika bralnega konca cevi, ustvari eno kopijo opisnika, na katero čaka, ko želi zasesti režo. Ko otrok umre, bo rokovalnik signala `SIGCHLD` zaprl kopijo, in s tem bo se bo vrnil iz sistemskega klica.

```
static int job_rfd = -1;

static int
make_job_rfd ()
{
  EINTRLOOP (job_rfd, dup (job_fds[0]));
  // ...
  return job_rfd;
}

void
jobserver_signal ()
{
  if (job_rfd >= 0)
    {
      close (job_rfd);
      job_rfd = -1;
```

[1] Na to omejitev sicer ne bi naleteli, če bi imeli proces z več nitmi, ampak je GNU Make zaradi prenosljivosti popolnoma eno-niten program.

```
    }
}

unsigned int
jobserver_acquire (int timeout)
{
  // ...
  EINTRLOOP (got_token, read (job_rfd, &intake, 1));
  // ...
}
```

Novejše različice Mac OS-a in Linuxa pa ponujajo sistemski klic, ki se imenuje `pselect`. `pselect` ponuja možnost čakanja na enega ali več datotečnih opisnikov in je lahko prekinjen s strani signala. Novejše različice GNU Make uporabljajo ta sistemski klic.

Na operacijskih sistemih Windows je pa jobserver implementiran popolnoma drugače. Od Windowsa XP naprej, Windows ponuja poimenovane semaforje (ang. *named semaphore*), ki predstavljajo atomski števec, do katerega lahko dostopajo vsi procesi na sistemu. Poimenovani semafor ustvarimo s funkcijo `CreateSemaphore`, pri čemer moramo podati ime, pod katerim bo semafor na voljo ostalim procesom [3].

```
unsigned int
jobserver_setup (int slots, const char *style)
{
  // ...
  sprintf (jobserver_semaphore_name, "gmake_semaphore_%d",
           _getpid ());

  jobserver_semaphore = CreateSemaphore (
      NULL,           /* Use default security descriptor */
      slots,                          /* Initial count */
      slots,                          /* Maximum count */
      jobserver_semaphore_name);      /* Semaphore name */

   // ...

   return 1;
}
```

Windows ponuja sistemsko funkcijo `WaitForMultipleObjects` iz knjižnice `Kernel32.dll`, podobno unixaški funkciji `select`, ki omogoča čakanje na več datotečnih opisnikov. Vendar, za razliko od `select`, ki podpira samo datotečne opisnike, `WaitForMultipleObjects` omogoča čakanje tudi na druge vire operacijskega sistema, vključno s procesi. Zaradi tega nimamo težav s hkratnim čakanjem na proces in semafor. [2]

[2]Starejše različice Windowsa sploh ne ponujajo mehanizma, ki bi bil enakovreden

```
static HANDLE jobserver_semaphore = NULL;

unsigned int
jobserver_setup (int slots, const char *style)
{
  sprintf (jobserver_semaphore_name, "gmake_semaphore_%d", _getpid ());

  jobserver_semaphore = CreateSemaphore (
      NULL,                           /* Use default security descriptor */
      slots,                          /* Initial count */
      slots,                          /* Maximum count */
      jobserver_semaphore_name);      /* Semaphore name */

  // ...
}

unsigned int
jobserver_acquire (int timeout UNUSED)
{
    HANDLE *handles;
    DWORD dwHandleCount;
    DWORD dwEvent;

    handles = xmalloc(process_table_actual_size() * sizeof(HANDLE));

    /* Add jobserver semaphore to first slot. */
    handles[0] = jobserver_semaphore;

    /* Build array of handles to wait for.  */
    dwHandleCount = 1 + process_set_handles (&handles[1]);

    dwEvent = process_wait_for_multiple_objects (
        dwHandleCount,  /* number of objects in array */
        handles,        /* array of objects */
        FALSE,          /* wait for any object */
        INFINITE);      /* wait until object is signalled */

    free(handles);

    // ...

    /* WAIT_OBJECT_0 indicates that the semaphore was signalled.
*/
    return dwEvent == WAIT_OBJECT_0;
```

unixaškemu signalu `SIGCHLD`, kjer bi OS proces obvestil o koncu otroškega procesa.

}

Literatura

[1] Association for Computing Machinery. Acm honors creator of landmark software tool, 2004. https://web.archive.org/web/20061213153355/http://www.acm.org/announcements/softwaresystemaward.3-24-04.html.

[2] Jagielski, J. Dmake, 2022. https://jimjag.github.io/dmake.

[3] Microsoft. Createsemaphorea function (winbase.h), 2022. https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea.

[4] Microsoft. Nmake reference, 2024. https://learn.microsoft.com/en-us/cpp/build/reference/nmake-reference.

[5] Miller, P. Recursive make considered harmful.

[6] Smith, P. D. Jobserver implementation. https://make.mad-scientist.net/papers/jobserver-implementation/.