Laboratorul 02. Tools

Până acum ați compilat programe pe același sistem pe care le-ați și rulat. Multe programe însă necesită un proces complex de compilare, care intern execută mulți pași și, în funcție de configurarea aleasă, poate dura un timp îndelungat. Pentru a reduce acest timp se dorește bineînțeles folosirea unei mașini cât mai puternice. În cazul sistemelor embedded este puțin probabil ca procesorul folosit să fie unul performant (în comparație cu un sistem desktop) sau ca memoria RAM disponibilă să fie mare. De altfel, nu de puține ori, avem de-a face cu un sistem care nici măcar nu are destul spațiu de stocare pentru sursele și fișierele binare generate în timpul compilării.

Pentru a elimina inconvenientele compilării pe sistemul embedded (**target**-ul) compilarea se face de obicei pe un sistem desktop (**host**-ul). Bineînțeles, acum pot apărea probleme dacă *target*-ul și *host*-ul folosesc procesoare cu arhitecturi diferite (executabilul generat de *host* nu va fi înțeles de procesorul *target*-ului). Aceste probleme apar deoarece compilarea va folosi în mod implicit compilatorul *host*-ului: *host-compiler*-ul (ex: gcc). Rezolvarea constă în instalarea pe *host* a unui compilator care poate genera executabile înțelese de *target*. Acest compilator poartă denumirea de **cross-compiler**, el rulând pe arhitectura *host*-ului, dar generând cod pentru arhitectura *target*-ului. Procesul prin care un program este compilat pe un alt sistem față de sistemul *target* se numește **cross-compilare**.

Această tehnică este folosită pentru a separa mediul de dezvoltare de mediul în care programul trebuie să ruleze. Acest lucru ne este util în cazul:

- sistemelor embedded, sisteme limitate din punct de vedere hardware, unde resursele nu sunt suficiente pentru întreg mediul de dezvoltare (ex: AVR, iOS, Android etc.);
- compilării pentru arhitecturi diferite, un exemplu fiind distribuțiile Linux, unde se poate folosi o singură mașină pentru a compila pentru diferite arhitecturi (ex: x86, x86-64, ARM etc.) kernelul și restul distribuției;
- compilării programului într-o fermă de servere, unde pentru performanță maximă, se va putea folosi orice mașină disponibilă, indiferent de arhitectura procesorului *host* sau a versiunii sistemului de operare.

Diferențierea între *host-compiler* și *cross-compiler* se face prin prefixarea acestuia din urmă cu un string, denumit **prefix**, care, prin convenție, conține o serie de informații despre arhitectura *target* (ex: `arm-bcm2708hardfp-linux-gnueabi-`). De asemenea, și restul utilitarelor folosite pentru compilare (ex: as, ld, objcopy etc.) vor avea același prefix. Tot ce trebuie să facem este să instruim sistemul de build să folosească *cross-complier*-ul pentru compilare.

Prefixul unui cross compiler se termină întotdeaduna cu `-`. El va fi concatenat la numele utilitarelor (ex: `gcc`) pentru a obține numele complet (ex: `arm-bcm2708hardfp-linux-gnueabi-gcc`)

Prin convenție, majoritatea sistemelor de build folosite în lumea Linux acceptă variabila de mediu `CROSS_COMPILE` pentru specificarea prefixului care trebuie folosit atunci când se dorește o *cross-compilare*.

Toolchain

Toolchain-ul reprezintă colecția de programe de dezvoltare software care sunt folosite pentru a compila și a obține un program executabil.

Programele care în majoritatea cazurilor sunt incluse în toolchain sunt:

- gcc - compilatorul de C;
- g++ - compilatorul de C++;
- as - assamblorul;
- ld - linker-ul;
- objcopy - copiază dintr-un fișier obiect în alt fișier obiect;
- objdump - afișează informații despre fișierul obiect;
- gdb - debugging.

make

Un alt program important pentru dezvoltarea unui sistem embedded, și nu numai, îl reprezintă *make*. Acest utilitar ne permite automatizarea și eficientizarea procesului de compilare prin intermediul fișierelor *Makefile*. Pentru o reamintire a modului de scriere a unui *Makefile* revedeți laboratorul de USO - Dezvoltarea programelor în C sub mediul Linux [http://ocw.cs.pub.ro/courses/uso/laboratoare/laborator-03].

Pentru ușurarea dezvoltării pe multiple sisteme embedded, fiecare având toolchain-ul lui propriu, vom dori să scriem *Makefile*-uri generice, care pot fi refolosite atunci când prefixul *cross-compiler*-ului se schimbă. Pentru aceasta va trebui să parametrizăm numele utilitarelor apelate în *Makefile*. Putem folosi în acest caz variabile de mediu în cadrul *Makefile*-ului. Acestea pot fi configurate apoi din exterior în funcție de sistemul *target* pentru care compilăm la un moment dat, fără a mai fi necesară editarea *Makefile*-urilor.

Cel mai simplu mod de a face acest lucru este să urmăm convenția deja stabilită pentru variabila de mediu care conține prefixul *cross-compiler*-ului: `CROSS_COMPILE`. Putem folosi această variabilă de mediu în cadrul *Makefile*-ului nostru utilizând sintaxa de expandare unei variabile, `$(<variabila>)`, și prefixând numele utilitarului cu variabila pentru prefix.

Makefile

```
hello: hello.c
	$(CROSS_COMPILE)gcc hello.c -o hello
```

Orice variabilă exportată în shell-ul curent va fi disponibilă și în fișierul *Makefile*. Putem de asemenea pasa variabile utilitarului *make* și sub formă de parametri, astfel:

```
$ make CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- hello
```

bash

O mare parte din dezvoltarea unui sistem embedded se face prin intermediul terminalului. Shell-ul care rulează în terminal permite personalizarea unor aspecte utile pentru dezvoltare precum variabilele de mediu încărcate la fiecare rulare. Aceste personalizări se fac însă în fișiere de configurare specifice fiecărui shell.

Pentru *bash* aceste fișiere reprezintă niste script-uri care sunt rulate automat și se găsesc în `/etc` (afectează toți utilizatorii) și în `$HOME` (afectează un singur utilizator). Prin intermediul fișierelor din `$HOME` fiecare utilizator își poate personaliza shell-urile pentru propriile nevoi. Aceste fișiere sunt:

- `.bash_profile` - este executat când se pornește un shell de login (ex: primul shell după logare);
- `.bashrc` - este executat cand se pornește orice shell interactiv (ex: orice terminal deschis);

- `.bash_logout` - este executat când shell-ul de login se închide.

Un alt fișier util folosit de *bash* este `.bash_history`, care memorează un istoric al comenzilor interactive rulate. Istoricul comenzilor este salvat în acest fișier la închiderea unui shell. Pentru o reamintire a unor comenzi utile în linia de comandă puteți revizita laboratorul de USO - Automatizare în linia de comandă [http://ocw.cs.pub.ro/courses/uso/laboratoare/laborator-06/].

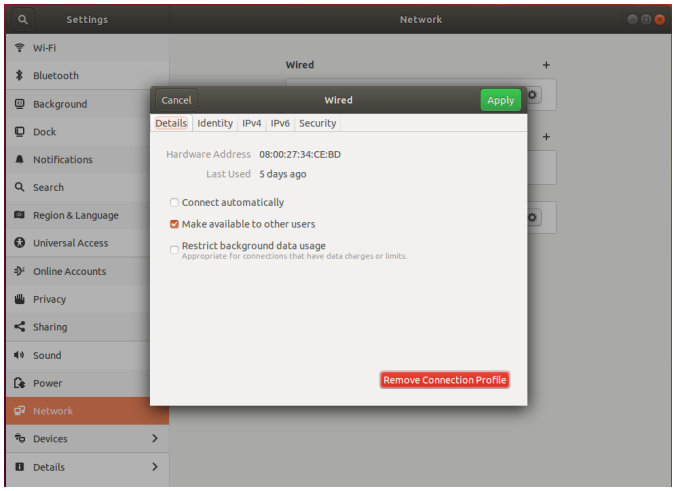
În dezvoltarea unui sistem embedded este deseori utilă adăugarea în variabila $PATH a căilor către diferitele tool-uri folosite, pentru ca acestea să poată fi accesate direct prin numele executabilului. Modificarea variabilei $PATH pentru fiecare shell pornit se poate face ușor prin intermediul fișierelor de personalizare a shell-ului.

Exerciții

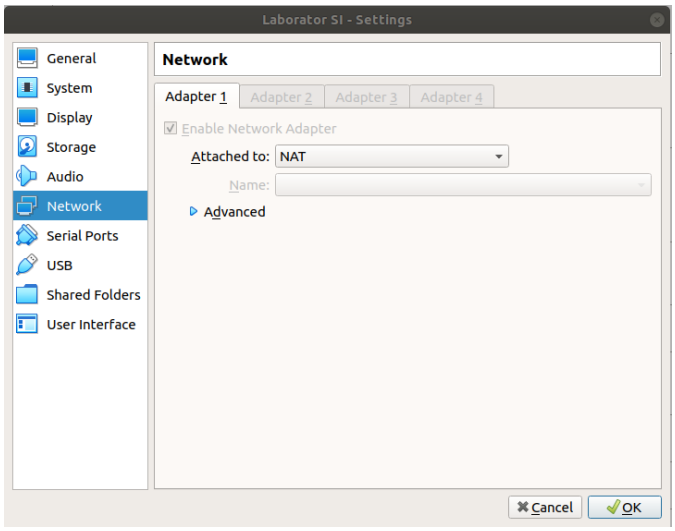
Atenție! Pentru rezolvarea laboratorului vom folosi aceeași mașină virtuală Ubuntu 18.04 Bionic Beaver pe care am folosit-o în **Laboratorul 1**, download-ată de pe osboxes [https://www.osboxes.org/ubuntu/] și rulată în VirtualBox [https://www.virtualbox.org/] sau VmWare. Recomandăm să folosiți **VirtualBox**!
Hint: ca să meargă copy-paste între host și guest pe VirtualBox, trebuie să activați Devices → Shared Clipboard → Bidirectional și să instalați Guest Additions (Devices → Insert Guest Additions CD Image).

0. Vom configura accesul *target*-ului la Internet.

Întâi vom dezactiva conectarea automată din setările sistemului de operare ale mașinii virtuale Ubuntu 18.04: Settings → Network → Wired → Connect Automatically (off).



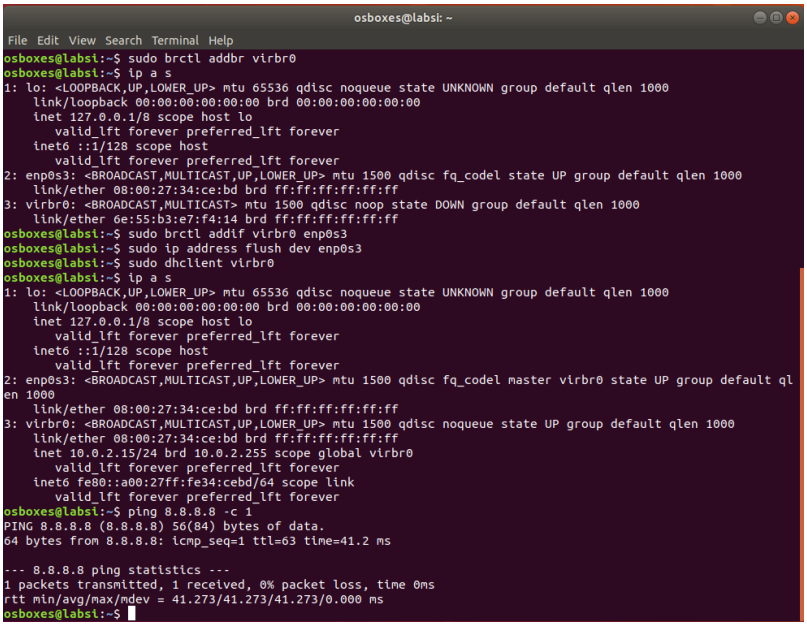
Închidem mașina virtuală. Înainte de a o porni, vom pune placa de rețea a mașinii virtuale Ubuntu in modul de NAT: Click dreapta pe mașina virtuală în VirtualBox → Settings → Network → Network Settings.



Pornim mașina virtuală (de acum la următoarele porniri, conexiunea la internet nu se va mai realiza automat). Vom crea *bridge*-ul **virbr0** care să ofere *target*-ului accesul la rețeaua fizică a *host*-ului:

```
sudo brctl addbr virbr0                                 # creăm bridge-ul
sudo brctl addif virbr0 <interfata fizica>              # adăugam interfața fizică a host-ului la bridge
sudo ip address flush dev <interfata fizica>            # ștergem adresa IP de pe interfața fizică, doar dacă avem o adresă
                                                        # IP pe interfață. Va șterge și ruta default automat
sudo dhclient virbr0                                    # obținem adresa IP pentru bridge și ruta default prin DHCP
```

Un exemplu de rulare ale comenzilor de deasupra:



Pentru ca *bridge*-ul să fie acceptat de QEMU el trebuie configurat și în fișierul `/etc/qemu/bridge.conf` sub forma:

bridge.conf

```
allow virbr0
```

Rulăm distribuția Raspbian folosind QEMU în modul *system emulation*:

- Kernel-ul modificat pentru a rula pe platforma Versatile PB este disponibil aici [https://drive.google.com/open?id=0B0lgiPZNMMyvaEtfN3V4VVBxRjg];
- Imaginea hard disk-ului (*rootfs*-ul) modificată pentru a rula pe platforma Versatile PB este disponibilă aici [https://drive.google.com/open?id=0B0lgiPZNMMyvOTFMakFuY1N2Q1E].

```
sudo qemu-system-arm -machine versatilepb -cpu arm1176 -kernel <kernel file> -append 'root=/dev/sda2' -drive file=<rootfs file>,index=0,media=disk,format=raw -net nic,model=smc91c111,netd
```

Logarea pe *target* se face cu credentialele:

- Username: **pi**
- Password: **raspberry**

Verificăm conectivitatea la internet:

```
ping 8.8.8.8
```

În continuare vom lucra cu *target*-ul pornit şi ne vom conecta la el prin SSH.

1. Ca să nu fiti nevoiţi să introduceţi de fiecare dată IP-ul *target*-ului, configuraţi pe *host* în **/etc/hosts** maparea dintre adresa IP a *target*-ului şi numele pe care doriţi să îl asociaţi (ex: **raspberry**).

- Puteţi afla adresa IP de pe *target* folosind comanda:

```
ip address show
```

- Puteţi verifica maparea făcută folosind ping următoare de pe *host*:

```
ping raspberry
```

- Mai multe detalii găsiţi aici [https://geek-university.com/linux/etc-hosts-file/]

2. Configuraţi conexiunea SSH către *target* astfel încât aceasta să se efectueze fără să mai fiţi nevoiţi să introduceţi parola de fiecare dată.

- Generaţi o cheie publică pe *host* (Hint: **ssh-keygen**)
- Copiaţi cheia publică generată pe *target* (Hint: **ssh-copy-id**)
- Puteţi verifica folosind comanda următoare de pe *host*:

```
ssh pi@raspberry
```

- Mai multe detalii găsiţi aici [https://www.thegeekstuff.com/2008/11/3-steps-to-perform-ssh-login-without-password-using-ssh-keygen-ssh-copy-id/]

3. Scrieți și compilați pe *host* un program **hello world**. Rulați acest executabil atât pe *host*, cât și pe *target*-ul RaspberyPi. Ce observaţi când rulaţi pe *target*?

- Folosiți `scp <sursă> <destinație>` pentru a copia executabilul pe *target*, unde `<sursă>` este calea către fișierul local (de pe *host*), iar `<destinație>` este calea remote către directorul de pe *target* unde se va salva (*<username>@<hostname>:<cale pe target>*). De exemplu:

```
scp hello pi@raspberry:.
```

4. Compilați program-ul **hello world** pentru a putea rula pe *target*-ul RaspberryPi. Salvați comanda folosită pentru compilare. Rulaţi programul compilat pe *target*.

- Instalați *toolchain*-ul pentru RaspberryPi (local, pe *host*), dacă acesta nu există deja, pentru a putea cross-compila programe pentru RaspberryPi:
  - Clonați/download-ați repo-ul [https://github.com/raspberrypi/tools];
  - Adăugați in $PATH calea către directorul ce conține executabilele necesare:

```
git clone --depth=1 https://github.com/raspberrypi/tools.git
export PATH="/home/osboxes/tools/arm-bcm2708/arm-linux-gnueabihf/bin:$PATH"
```

- Prefixul *cross-compiler*-ului este `arm-linux-gnueabihf-`. Compilatorul este `arm-linux-gnueabihf-gcc`.

5. Creați un *Makefile* generic pentru programul **hello world** care poate compila pentru orice sistem *target* în funcție de variabilele primite (convenția `CROSS_COMPILE`). Compilați programul pentru *host* și pentru *target*-ul RaspberryPi, apoi salvați executabilele generate.

- Dacă o variabilă nu este setată, construcția `$(<variabilă>)` într-un *Makefile* va fi echivalentă cu șirul vid.

6. Ce puteți spune despre conținutul celor 2 fișiere executabile create la exercițiul anterior?

- Pentru informaţii legate de tipul fişierelor se poate folosi comanda **file**;
- Conținutul unui fișier executabil poate fi inspectat cu utilitarul **objdump** (ptr *target* folosiţi utilitarul din toolchain: **arm-linux-gnueabihf-objdump**)

Resurse

- Soluție laborator
- Kernel Raspbian compatibil cu QEMU [https://drive.google.com/open?id=0B0IgiPZNMMyvaEtfN3V4VVBxRjg]
- Rootfs Raspbian compatibil cu QEMU [https://drive.google.com/open?id=0B0IgiPZNMMyvOTFMakFuY1N2Q1E]
- Configurarea fişierului /etc/hosts [https://geek-university.com/linux/etc-hosts-file/]
- 3 paşi pentru autentificare prin SSH fără parolă folosind ssh-keygen şi ssh-copy-id [https://www.thegeekstuff.com/2008/11/3-steps-to-perform-ssh-login-without-password-using-ssh-keygen-ssh-copy-id/]
- Automatizare în linia de comandă [http://ocw.cs.pub.ro/courses/uso/laboratoare/laborator-06] - USO
- Dezvoltarea programelor în C sub mediul Linux [http://ocw.cs.pub.ro/courses/uso/laboratoare/laborator-03] - USO

si/laboratoare/02.txt · Last modified: 2020/10/17 09:40 by stefan_radu.maftei