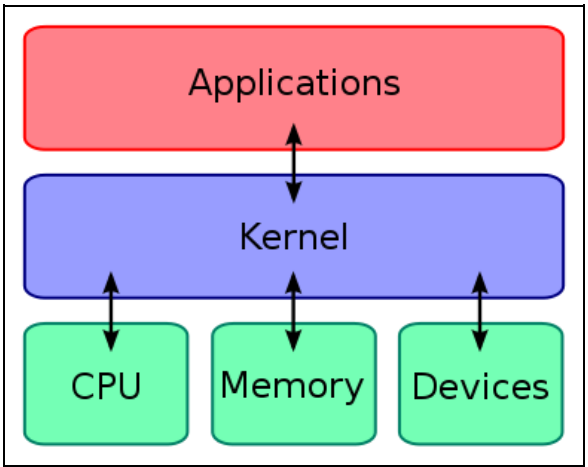
Laboratorul 03. Kernel

Kernel-ul reprezintă o parte a sistemului de operare responsabilă cu accesul la hardware și managementul dispozitivelor dintr-un sistem de calcul (ex: procesoul, memoria, dispozitivele de I/O). De asemenea, el are rolul de a simplifica accesul la diferitele dispozitive hardware, oferind o interfață generică pentru aplicații prin intermediul system-call-urilor. În spatele interfeței generice se află porțiuni din kernel, numite drivere, care implementeză comunicația cu dispozitivele hardware. Un alt rol al kernel-ului este de a izola aplicațiile între ele, atât pentru stabilitatea sistemului, cât și din considerente de securitate.



Architectura unui sistem de operare

Pentru a îndeplini toate aceste sarcini, codul kernel-ului rulează într-un mod special de lucru al procesorului, fapt care îi permite să execute o serie de instrucțiuni privilegiate. Acest mod privilegiat de lucru nu este accesibil aplicațiilor obișnuite. Spunem că aplicațiile rulează în *user-space* (modul neprivilegiat), iar kernel-ul rulează în *kernel-space* (modul privilegiat).

Linux

Linux este numele unui kernel creat de către Linus Torvalds, care stă la baza tuturor distribuțiilor GNU/Linux. Inițial, el a fost scris pentru procesorul Intel 80386 însă, datorită licenței permisibile, a cunoscut o dezvoltare extraordinară, în ziua de astăzi el rulând pe o gamă largă de dispozitive, de la ceasuri de mână până la super-calculatoare. Această versatilitate, cât și numărul mare de arhitecturi și de periferice suportate, îl face ideal ca bază pentru un sistem embedded.

Arhitecturile suportate de kernel-ul Linux se pot afla listând conținutul directorului `arch` din cadrul surselor.

Linux este un kernel cu o arhitectură monolitică, acest lucru însemnând că toate serviciile oferite de kernel rulează în același spațiu de adresă și cu aceleași privilegii. Linux permite însă și încărcarea dinamică de cod (în timpul execuției) în kernel prin intermediul modulelor. Astfel, putem avea disponibile o multime de drivere, însă cele care nu sunt folosite des nu vor fi încarcate și nu vor rula. Spre deosebire de aplicații însă, care rulează în modul neprivilegiat (*user-space*) și nu pot afecta funcționarea kernel-ului, un modul are acces la toată memoria kernel-ului și se execută în *kernel-space* (poate executa orice instrucțiune privilegiată). Un bug într-un modul sau un modul malițios poate compromite întregul sistem.

Dezvoltarea kernel-ului Linux se face în mod distribuit, folosind sistemul de versionare Git. Versiunea oficială a kernel-ului, denumită *mainline* sau *vanilla* este disponibilă în repository-ul lui Linus Torvalds, la adresa https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git [https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git].

Versiunea oficială este însă rar folosită într-un sistem embedded nemodificată. Este foarte comun ca fiecare sistem să folosească o versiune proprie a kernelului (numită un *tree*) bazată mai mult sau mai puțin pe versiunea oficială. Datorită licenței GPLv2 a kernelului, însă, orice producător care folosește o versiune modificată a kernelului este obligat să pună la dispoziție modificările aduse.

Aceste modificări sunt puse la dispoziție sub formă de *patch*-uri care trebuie aplicate unei anumite versiuni de kernel. O altă modalitate, care este folosită și de către fundația RaspberryPi, este de a publica un repository de

Git cu versiunea modificată (un *tree* alternativ). Datorită modelului distribuit de dezvoltare suportat de Git, această a doua metodă are avantajul că permite dezvoltarea ușoară în paralel a celor două versiuni. Modificările făcute într-una pot fi portate și în cealaltă, iar Git va ține minte ce diferențe există în fiecare versiune. Cele două versiuni sunt de fapt două *branch*-uri de dezvoltare, care se întâmplă să fie găzduite pe servere diferite.

Kernel-ul folosit pe RaspberryPi, care include suportul pentru SoC-ul Broadcom BCM2385 folosit de acesta, se găsește la adresa https://github.com/raspberrypi/linux.git [https://github.com/raspberrypi/linux.git].

Linux kernel build system

Pentru compilare și generarea tuturor componentelor kernel-ului (ex: imaginea principală - *vmlinux*, module, firmware) Linux folosește un sistem de build dezvoltat o dată cu kernel-ul, bazat pe utilitarul *make*. Acest sistem de build însă nu seamănă cu clasicul *config/make/make install*, deși ambele sunt bazate pe utilitarul *make*.

Toți pașii de compilare sunt implementați ca target-uri pentru *make*. De exemplu, pentru configurare se poate folosi target-ul `config`. Această metodă de configurare însă nu este recomandată deoarece oferă o interfață foarte greoaie de configurare a kernel-ului.

Target-ul `help` oferă informații despre aproape toate operațiile suportate de către sistemul de build.

```
$ make help
```

Operațiile oferite sunt grupate în diferite categorii:

- *cleaning* - conține target-uri pentru ștergerea fișierelor generate la compilare
  - spre deosebire de `clean`, `mrproper` șterge în plus toate fișierele generate, plus configurarea și diferite fișiere de backup
- *configuration* - conține diferite target-uri pentru generarea unei configurări a kernelului; există target-uri care permit:
  - generarea unui fișier de configurare nou; ex: `defconfig`, `allmodconfig`, …
  - actualizarea unui fișier de configurare existent; ex: `olddefconfig`, `localyesconfig`, …
  - editarea unui fișier de configurare existent; ex: `menuconfig`, `nconfig`, …
- *generic* - conține target-uri generice; există target-uri pentru:
  - compilare; ex: `all` - target-ul implicit care este executat la o invocare simplă a lui `make`, iar `vmlinux` și `modules` compilează imaginea kernel-ului și, respectiv, modulele selectate
  - instalare; ex: `headers_install`, `module_install`, …
  - informații; ex: `kernelrelease`, `image_name`, …
- *architecture dependent* - conține target-uri dependente de architectură, care diferă în funcție de arhitectura selectată; există target-uri pentru;
  - generarea de imagini în diferite formate: compresate (`zImage` și `bzImage`), pentru U-Boot (`uImage`), …
  - generarea de configurări implicite pentru sisteme bazate pe arhitectura selectată; ex: `*_defconfig`

Arhitectura care va fi compilată este selectată de variabila de mediu ARCH.

```
$ ARCH=x86 make [<targets>]
sau
$ make ARCH=x86 [<targets>]
```

Dacă arhitectura nu este setată explicit, se folosește implicit arhitectura sistemului pe care se face compilarea.

În cele mai multe situații se dorește compilarea unui kernel pentru un sistem deja existent, fie pentru a adăuga sau elimina funcționalități sau pentru a actualiza versiunea de kernel folosită. În aceste cazuri folosirea target-urilor de generare a unei configurații noi, chiar și a celor care generează o configurație implicită pentru arhitectura noastră nu sunt neapărat utile. Este posibil ca kernel-ul existent să aibă deja o configurație personalizată care se dorește doar a fi actualizată/modificată folosind target-urile de editare.

Un kernel care rulează poate conține fișierul de configurare (de obicei în format comprimat *gzip*) din care a fost compilat, dacă această funcționalitatea a fost selectată la build. Acest fișier se regăsește în `/proc/config.gz`. Tot ce rămâne este să extragem acest fișier și să-l modificăm conform dorințelor.

Testare

Pentru a testa un nou kernel acesta trebuie instalat pe *target*. Această procedură diferă de la sistem la sistem, iar pe RaspberryPi constă în copierea acestuia pe card-ul SD în partiția de *boot* sub numele de `kernel.img`. În momentul dezvoltării și testării unui nou kernel, instalarea fiecărei versiuni a acestuia pe *target* reprezintă un bottleneck major.

O alternativă la instalarea kernel-ului pe *target* o reprezintă încărcarea acestuia prin rețea direct pe de *host*-ul folosit la dezvoltare, dacă există suport din partea bootloader-ului. Din păcate, bootloader-ul implicit de pe RaspberryPi nu are suport pentru a încărca o imagine de kernel de pe rețea. Un bootloader care oferă însă acestă facilitate este *U-Boot* [2], el folosind protocolul TFTP pentru a boota o imagine de kernel prin rețea.

Instalare pachete/programe din surse

De multe ori ne lovim de problema instalării unui pachet sau a unui program pe care îl găsim doar pe un repository public, de cele mai multe ori bazat pe Git. Astfel, pentru a ne putea folosi de acel pachet/program, trebuie să cunoaștem următoarele utilitare:

Git

Opțiuni și comenzi git:

1. `git clone <repo>` - va aduce toate fișierele conținute de repository-ul *repo* pe mașina locală.
2. `git pull` - actualizează un repository deja clonat local la ultima versiune remote.
3. `git checkout <branch>` sau `git checkout <tag>` - actualizează fișierele locale la versiunea indicată de *branch* sau de *tag*. Un repository poate avea mai multe branch-uri, care pot fi văzute ca niște versiuni diferite ale repository-ului. Un exemplu uzual de folosire a branch-urilor este pentru organizarea diferitelor versiuni ale aceluiași program.
4. `git apply <patch file>` - aplică pe fișierele locale modificările conținute de fișierul *patch*.

diff și patch

diff este un utilitar cu care se pot afișa diferențele dintre două fișiere. De altfel, el poate fi folosit și pentru a genera un fișier *patch* cu acele diferențe. Un fișier *patch* descrie modificările care trebuie făcute asupra unui fișier sau director pentru a se ajunge dintr-o starea A într-o altă stare B. În lumea Linux există două modalități mai importante de a genera un *patch*:

1. utilitarul **diff**: `diff <path A> <path B>`
2. utilitarul **Git**: `git diff <path>` - afișează diferențele dintre versiunea salvată pe disc (starea B) și versiunea nemodificată, memorată de repository (starea A)

și respectiv două modalități de a aplica acest *patch*:

1. utilitarul **patch**: `patch -p1 < <patch file>` - va aplica modificările descrise de fișierul *patch* în directorul curent
2. utilitarul **Git**: `git apply <patch_file>`

Exerciții

1. Listați denumirile configurărilor implicite (*defconfig*) disponibile în ultima versiune a kernel-ului RaspberryPi pentru platformele Versatile PB și BCM. Generați fișierul de configurare pentru una dintre aceste configurații. Studiați 2min formatul acestui fișier si reflectați asupra acestuia îndelung.

- Clonați repository-ul de git al kernel-ului RaspberryPi. Acesta este disponibil la adresa https://github.com/raspberrypi/linux.git [https://github.com/raspberrypi/linux.git].
- Asigurați-vă că aveți instalat pachetele `bison` și `flex`.
- Folosiți *help*-ul sistemului de build pentru a afla denumirile configurațiilor și pentru a genera fișierul de configurare.
- Deschideți fișierul `.config` cu un editor de text pentru a vedea conținutul configurării.

Pentru a reduce timpul de download, puteti descarca doar o anumita versiune de kernel

- mkdir linux
- cd linux
- git init
- git remote add origin <adresa repo>
- git fetch –depth 1 origin <versiune>
  - eg: git fetch –depth 1 origin rpi-3.18.y
- git checkout rpi-3.18.y

Pentru a putea vedea alta versiune, trebuie facut fetch pentru ea, in mod similar

- Revedeți despre *git*.
- Revedeți modul de folosire a sistemului de build.
- **grep** și operatorul de shell | (*pipe*) sunt prietenii voștri.
- Nu uitați că arhitectura celor două platforme este *ARM*. Variabila de mediu ARCH trebuie setată pentru toate invocările *make*. Altfel veți utiliza arhitectura host-ului (x86), cu rezultate derutante.
- Revedeți laboratorul [http://ocw.cs.pub.ro/courses/uso/laboratoare/laborator-06] de USO despre folosirea linei de comandă.

2. Compilați și rulați în QEMU ultima versiune a kernel-ului disponibil pe Raspbian Wheezy, urmărind instrucțiunile de mai jos:

După cum ați văzut în laboratorul 1, pentru a rula distribuția Raspbian Wheezy [https://www.raspberrypi.org/downloads/raspbian/] în QEMU vom folosi platforma Versatile PB [http://infocenter.arm.com/help/topic/com.arm.doc.dui0225d/DUI0225D_versatile_application_baseboard_arm926ej_s_ug.pdf] cu un procesor ARM1176JZ-F [http://www.arm.com/products/processors/classic/arm11/arm1176.php], cu arhitectura ARMv6.

Daca se doreste compilarea fara configurare manuala ci doar cu aplicare de patchuri se recomanda parcurgerea pasilor de mai jos citind totodata continutul patch-urilor:

- Schimbați repository-ul pe *branch-ul* `rpi-3.18.y` pentru a obține sursele kernel-ului folosit în ultimul Raspbian Wheezy.
- Configurați corespunzător kernel-ul.
  - Aplicați patch-ul pentru a permite arhitecturii ARMv6 să fie compilată într-un kernel pentru Versatile PB.
  - Generați fișierul de configurare implicită pentru platforma Versatile PB.
  - Aplicați patch-ul pentru a modifica automat configurarea, precum descrierii de mai sus.
- Compilați kernel-ul.
- Rulați kernel-ul în QEMU, folosind aceeași metodă utilizată și în laboratorul 1. *Rootfs*-ul necesar este disponibil aici [https://drive.google.com/open?id=0B0lgiPZNMMyvOTFMakFuY1N2Q1E].
- Afișați versiunea kernel-ului care rulează folosind comanda `uname -a`.

- Revedeți secțiunea despre *patch*.
- Folosiți variabila de mediu `CROSS_COMPILE` pentru a configura prefixul compilatorului folosit pentru compilare.
- Folosiți parametrul `-j 4` al *make* pentru a paraleliza compilarea pe 4 procese (host-urile au procesoare dual-core).
- Compilarea durează aproximativ 6min.

- Imaginea rezultată se găsește în `arch/arm/boot/zImage`

Daca ați clonat repository-ul de pe un calculator local, este posibil să nu găsiți anumite branchuri. Puteti reseta originea pentru git folosind urmatoarele comenzi:

- `git remote remove origin`
- `git remote add origin https://github.com/raspberrypi/linux.git` `[https://github.com/raspberrypi/linux.git]`
- `git fetch`

Explicatia celor două patch-uri de mai sus:

Pentru a putea rula însă kernel-ul 3.18 pentru RaspberryPi pe această configurație va trebui să-i aducem câteva modificări.

1 Prima este de a activa suportul pentru arhitectura ARMv6 pe platforma Versatile PB, kernel-ul original nesuportând această arthitectură pe platforma noastră.

2. În continuare vom porni de la configurația implicită a kernel-ului pentru platforma Versatile PB și vom adăuga câteva feature-uri și drivere (opțiunile **nu** trebuie compilate sub formă de module; dorim <*>, nu <M>) pentru a putea boota *rootfs*-ul Raspbian Wheezy, după cum urmează:

- pentru a configura arhitectura ARMv6 activați:
  - System Type →
    - Support ARM V6 processor
    - ARM errata: Invalidation of the Instruction Cache operation can fail
    - ARM errata: Possible cache data corruption with hit-under-miss enabled
- pentru a include driver-ul pentru bus-ul PCI folosit de controller-ul de hard disk activați:
  - Bus support →
    - PCI support
- pentru a include driver-ul pentru controller-ul de hard disk activați:
  - Device Drivers →
    - SCSI device support →
      - SCSI device support
      - SCSI disk support
      - SCSI low-level drivers →
        - SYM53C8XX Version 2 SCSI support
- pentru a include driver-ul pentru sistemul de fișiere Ext 4 activați:
  - File systems →
    - The Extended 4 (ext4) filesystem
- pentru a include pseudo-sistemul de fișiere *tmpfs* necesar funcționării distribuției activați:
  - File systems →
    - Pseudo filesystems →
      - Tmpfs virtual memory file system support (former shm fs)
- pentru a suporta binarele din distribuția Raspbian activați:
  - Floating point emulation →
    - VFP-format floating point maths
- pentru a repara o eroare de compilare în această configurație **dezactivați**:
  - Device Drivers →
    - MMC/SD/SDIO card support

- Pentru a realiza configuratiile (Pasul 2) de mai sus manual (fara a aplica cel de-al doilea patch)se recomanda folosirea meniului interactiv: make menuconfig
- Utilizati “Space” pentru modificare configuratii si ”/” pentru find.
- make menuconfig are nevoie de suport aditional de biblioteci: libncurses5-dev si libncursesw5-dev (! Instalati folosind package installer)

Resurse

- Soluție laborator (disponibilă mai târziu)
- Rootfs Raspbian Wheezy compatibil cu QEMU [https://drive.google.com/open?id=0B0lgiPZNMMyvOTFMakFuY1N2Q1E]
- Kernel Raspbian Wheezy original [https://drive.google.com/open?id=0B0lgiPZNMMyvUlEwcUhHenp4MjA]
- Kernel patch pentru activarea arhitecturii ARMv6 pentru platforma Versatile PB
- Kernel patch pentru a putea rula configurația implicită a Versatile PB în QEMU

Referințe

1. Ghid compilare kernel RaspberryPi [http://elinux.org/Raspberry_Pi_Kernel_Compilation]
2. Ghid compilare U-Boot pentru RaspberryPi [http://elinux.org/RPi_U-Boot]

si/laboratoare/03.txt · Last modified: 2020/10/27 22:41 by laura.ruse