Embedding BSD

1. **Въведение**

Настоящото изложение е насочено към разглеждане плюсовете и минусите при използване на операционната система BSD в embedded проекти. Тематиката обхваща разглеждане на вътрешната архитектура на операционната система, процеса на „движение" на данни между отделни части, възможностите и начините за интерференция на системен или приложен програмист, както и самия процес при изграждане на embedded builds.

2. **Целена Аудитория**

Материалът е предназначен за потребители с основни познания по BSD операционните системи. Допълнителни познания по хардуерната организация на компютъра, езика за програмиране C и shell scripting също са от полза.

3. **Въведение в embedded системите**

С бързото развитие на компютърния хардуер, поевтиняване на компонентите и високата интеграция, както и в следствие нуждите на индустрията се обособява един клон в (ако можем да се изразим така) компютърната наука, който може условно да бъде наречен „Embedded Systems". Embedded System/Appliances са всички онези компютърни системи, които са проектирани с цел извършване на една или няколко точно определени дейности, но извършвайки ги добре. Според определението в популярната Wikipedia "embedded system" е **„компютърна система със специално предназначение, изцяло енкапсулирана в устройството, което контролира, има специфични изисквания и изпълнява предварително дефинирани задачи"** ... за разлика от персоналните компютри с обща употреба. От това определение могат да бъдат извадени следните характеристики:

- **специално предназначение**
- **дълбока степен на интеграция между хардуер и софтуер**
- **специфични изисквания на/към средата**
- **предварително дефиниран набор от функции**

Специалното предназначение обхваща някакво множество от функции, към извършването на които е насочен дизайнът на съответната компютърна система. Съществуват различни видове вградени системи и тук ще бъдат посочени само няколко примера: мрежови устройства – рутери, принтери, сървъри, устройства за управление на процеси в домакинството, индустриални компютърни системи за контрол на производствени процеси и пр.

Дълбоката степен на интеграция между софтуер и хардуер трябва да се разглежда като специфичен дизайн на софтуерния комплекс, обслужващ съответната embedded система, с цел пълно използване на предложената от хардуера производителност или специфична функционалност. В определен брой случаи това изключва наличието на операционна система и се свежда до един контролен цикъл, изпълняващ последователност от действия при настъпване на вътрешни или външни събития.

Специфичните изисквания на/към средата са условията, в който ще работи или от които зависи правилното функциониране вградената

компютърна система. Често те са крайно неблагоприятни – проблем, с който се борят както хардуерните, така и софтуерните проектанти. Това обуславя и широкото използване на системи за мониторинг при дизайна на embedded системи. От друга страна, самата embedded система може да поставя специфични условия към своята оперативна среда.

Предварително дефинираният набор от функции е приложния интерфейс на системата. Това обхваща предварителен, изчистен, пълен набор от извиквания към системни процедури, които компютърната система трябва да предложи на изпълняваните в нея приложения. Проблемът може да се разглежда както от хардуерна, така и от софтуерна гледна точка. Програмният интерфейс е пряко свързан и с потребителския интерфейс, доколкото последният е част от първия.

Това кратко въведение в embedded системите дава основните насоки, на които трябва да се обърне внимание при сравнителни характеристики и оценка на системен софтуер. Това са и насоките, на които ще бъде обърнато внимание при оценяване на BSD операционните системи като възможни операционни среди при изграждане на embedded системи.

4. **BSD Architecture – Design Overview**

Архитектурата на BSD операционните системи е концептуално опростена и се подчинява на това, което днес е известно като „монолитен дизайн“. Операционната система е изградена около монолитно ядро и структура от процеси, обслужващи различна функционалност.

BSD ядрото е с монолитен дизайн (с изключение може би на бъдещите версии на DragonFlyBSD :)). Всички драйвери на устройства, софтуер за поддръжка на различни файлови системи, протоколни стекове и много други са част от самото ядро и работят в привилегирован режим. Кодът на ядрото може условно да бъде разделен на две части – системно-зависима и системно-независима.

Системно-зависимата обхваща частта, която обслужва съответната компютърна архитектура на ниско ниво и драйверите за директен достъп до хардуера.

Системно-независимата реализира останалата част от драйверите, необходими за обслужване на устройствата, поддръжката на множество файлови системи, междупроцесната комуникация (от части), мрежовия стек (протоколните стекове на различните протоколни реализации) и системния приложен интерфейс.

Разделянето на функционалността на системно-зависима и системно-независима е извършено с цел подобряване логическата организация на кода и улесняване процеса по пренасяне функционалността на ядрото на различни процесорни архитектури. Доказателство за успеха на този подход е NetBSD – операционна система, поддържаща над 50 различни процесорни архитектури.

От друга страна, това условно разделяне на кода на ядрото налага унифициран интерфейс при създаване на драйвери за устройства или файлови системи, позволява лесно добавяне или променяне на функционалност по отделни части на ядрото, което от своя страна улеснява системните програмисти, предоставяйки им завършен модел за комуникация между отделните части на ядрото,

функционално и логическо разделение на отделни елементи и не на последно място – прегледност на кода.

Системно-независимата част предоставя на приложния софтуер унифициран интерфейс за достъп до услугите, предлагани от ядрото – т.нар. syscalls. Това позволява разработване на приложен софтуер, независим от евентуални промени в кода на ядрото, не засягащи формите на комуникация с външни обекти.

От изложеното личи, че въпреки монолитния дизайн, организацията на BSD ядрото предлага завършен модел на взаимодействие на отделните части на ядрото както помежду си, така и между ядрото и приложния софтуер. Логическото разделение на функционалността е в основата на високата преносимост на кода и възможностите за бърза промяна или добавяне на системна функционалност.

Нивото на абстракция, следващо системно-независимата част от модела на операционната система е системната библиотека. Тя позволява на приложенията достъп до системната функционалност от по-високо ниво. Системната библиотека е „компромисът" между онова, което е излишно при имплементиране на приложния интерфейс на ядрото и абсолютно необходимо на приложния програмист. Така системната библиотека създава един относителен баланс между функционалност от високо ниво и достъп до системни ресурси предоставени от „по-ниския" слой на ядрото.

Най-високо място в така разглежданата йерархия заема приложеният софтуер. Това е програмното обезпечение, с което в повечето случай работи потребителят. Свободата при имплементиране на това ниво е напълно неограничена.

Разгледаният до момента материал позволява да се направят следните изводи за приложимостта на BSD архитектурата при изграждане на embedded системи:

1) BSD операционната система е оперативна среда, създадена с цел обслужване и предоставяне на функционалност с общо предназначение. Не съществуват (или поне не са широко разпространени и развити) специфични за вградените системи алгоритми и конструкции. Системната архитектура, обаче позволява създаването на такива без загуба на функционалност или изисквания за сериозни промени.

2) Двуслойният модел на ядрото предоставя възможност за имплементиране (и за поддържаните архитектури това е реализирано) на дълбока степен на интеграция между хардуер и софтуер. Унифицираният интерфейс между двете части позволява лесно създаване на високо производителни драйвери, използващи максимума от функционалност, предложена от наличния хардуер.

3) BSD операционната система не изисква специфични условия за правилна работа към оперативната среда. Модулният дизайн позволява да се заобиколят липси или недостатъци на хардуерния дизайн и високата портабилност го доказва (липси на MMU модули, human interface модули, и пр). От друга страна, операционната система може лесно да бъде пригодена да отговаря на специфични условия, поставени от приложен софтуер или специализиран хардуер към нея. И тук гъвкавостта на дизайна позволява бърза промяна и нагласа към специфичните условия на средата.

4)Системният приложен интерфейс (API) на ядрото както и

системната библиотека предлагат изчистен и стабилен достъп на приложения или системния софтуер до функционалност от „по-ниско“ ниво. Това създава предварително дефинирания набор от функции, изисквани от системата и е в основата на изграждане на модела, обслужващ предварително дефинирания набор от функции, изискван от embedded системата като завършен компонент.

Става ясно, че от гледна точка на системен дизайн BSD операционната система до голяма степен е подходяща за използване в embedded системи, доколкото успешно покрива голяма част от изискванията за признаване на една embedded система като такава.

5. **BSD Architecture – From POST to the Shell Prompt**

Следващият „разрез“, който трябва да бъде разгледан, е организацията на процеса по стартиране, зареждане и работа на операционната система. Този процес е относително праволинеен и обхваща следната последователност от действия. Поради възможни различия между различни BSD дистрибуции ще бъде разгледан само един концептуален модел.

След стартиране на компютърната система и преминаване процеса на инициализация на отделните устройства, обикновено се пристъпва към зареждане на операционна система от някаква външна памет – твърд диск, flash памет или др. Управлението се предава на кратка програма, boot loader, която съдържа код, достатъчен за локализиране, зареждане и предаване управлението на ядрото.

Ядрото проверява наличния хардуер, инициализира го, създава и попълва системни таблици, инициализира системни броячи, създава файлове на устройства, попълва /dev и /proc и/или /kern файловите системи. Инициализират се мрежовия стек и пакетните филтри. Тъй като вече са заредени драйверите за поддръжка на различните файлови системи, се монтира файловата система / (root) и се предава управлението на процеса /sbin/init.

Процесът /sbin/init (PID=1) стартира системата за runtime configurations (/etc/rc). Тя съдържа допълнителните системни настройки – инициализация на интерфейси, стартиране на допълнителни сървъри, демони и др. настройка на пакетни филтри, системни променливи и пр. Следва инициализация на наличните терминални устройства посредством стартиране на процес getty за всяко от тях със съответни параметри в зависимост от вида на самото устройство. В общия случай всеки getty процес стартира /bin/login. Към този момент системата е готова за работа с потребителя.

Очевидно, модулният подход присъства и тук. Всеки модул се грижи да подготви функционално определена част от оперативната среда. Има ясна последователност в процеса на действия. Пътят на предаване управлението между модулите е известен и стабилен. Допълнително предимство е и избраният метод за конфигуриране поведението на отделните процеси – това са текстови файлове. Цялата runtime configuration система е изградена от текстови файлове във shell script формат. Така на потребителя се осигурява лесен начин за интерференция в процеса на стартиране и конфигуриране на системата.

От друга страна, връзките между отделните модули са поименни. Това прави възможно съществуването на произволен код с име

/sbin/init, на който ще бъде предадено управлението от ядрото. По сходен начин може да се промени и /etc/rc, както и останалите разгледани по-горе процеси или последователности.

Достъпът до кода на операционната система предоставя съвсем различно ниво на модифициране на процеса на първоначално конфигуриране и стартиране. Възможността за редактиране кода на ядрото позволява цялостна промяна на пътя, по който стартира системата. Промени в /sbin/init или /etc/rc също са възможни и предлагат гъвкав механизъм за напасване към изискванията на работната среда. Възможността за намеса в работата на getty/login било чрез конфигурационни файлове, било чрез промяна в изходния код внася още едно ниво на евентуални промени.

6. **BSD за Embedded Systems. Плюсове (+) и Минуси (-).**

Изхождайки от горното изложение могат да бъдат изведени следните плюсове и минуси.

6.1. **Плюсове:**

- Модулна архитектура на ОС
- Множество подсистеми (FS, TCP/IP, VM, NETGRAPH, IPFW/IPF/PF, Audio, X, etc.)
- Ниски изисквания към наличен хардуер
- Голям брой поддържани процесорни архитектури
- Драйвери за множество устройства
- Стандартизирано API (SUSv3, POSIX)
- Отворен код на операционната система
- Множество налични приложения
- „Специализация" на различните BSD дистрибуции
- Качествен код
- Възможност за произволна промяна на кода
- Лиценз, позволяващ използване и в комерсиални проекти
- Съществуващи проекти, използващи BSD като операционна система в embedded enviroment
- Наличие на специално разработени tools (tinyware)

6.2. **Минуси:**

- Монолитна архитектура на ядрото
- Слаба поддръжка от страна на производителите на хардуер
- Липса на системна библиотека подходяща за използване в embedded проекти (dietlibc, uClibc)
- Липса на специализирани build tools (CrossTools, BuildRoot, BusyBox)

7. **Изграждане на Embedded BSD Builds**

Съществуват два подхода при изграждане на embedded builds на BSD операционни системи. Всеки от тях има предимства и недостатъци.

Основната разлика между тях е в това дали се ползват статично или динамично свързани изпълними файлове и дали използваната файлова система се намира във файла на ядрото (откъдето се зарежда като ramdisk) или извън него. Наличния обем дискова и оперативна памет също предопределя избора на единия или другия подход. Последващото изложение ще се опита да разгледа двата модела в

детайли.

Общото в двата модела е това, че се създава „умалено" копие на операционната система, което може да бъде стартирано посредством хардуера, за който е предназначено и съдържа файлова система с нужните за работата на устройството ядро, допълнителни помагала и конфигурационни файлове.

7.1 **Използване на динамично свързани изпълними файлове**

Създаденият embedded BSD build до голяма степен следва организацията на операционната система, на която е базиран. Изпълнимите файлове са динамично свързани и изискваните от тях библиотеки са част от файловата система на build-а. Използването на този подход е подходящо при наличие на устройство за съхранение на информация наподобяващо твърд диск. Това би могло да бъде както малък твърд диск, така и CD-ROM, така и енергонезависима памет (CompactFlash, USB Stick, etc). Изискване е операционната система да може да бъде заредена от него по някакъв начин, пряко устройството или чрез използване на boot loader. Допълнително изискване е и операционната система да има поддръжка за съответния тип устройство. Смятайки, че това е изпълнено, ще бъде разгледан пример, в който като външна памет се използва CompactFlash карта или малък диск.

Изисквания към средата:

- Операционна Система: FreeBSD 5-STABLE
- Необходими:
  - source кодът на операционната система
  - Perl интерпретатор
  - Bochs / Qemu

Изграждане на embedded BSD build:

Процесът се свежда до създаване „умалено" копие на host операционна система и трансферирането му върху устройството, от което ще бъде стартирана embedded системата.

1) Създаване на работна директория:

mkdir /core

2) Създаване на файл с характеристики, възможно най-близки до външния носител. Използват се помагала като **bximage** (от пакета Bochs) или предоставеният от операционната система **dd.**

3) Използва се възможността на FreeBSD да третира файлове като устройства:

mdconfig -a -t vnode -f drive.img -u md0

4) Реинициализира таблицата на устройствата и се създава един FreeBSD дял, който използва максимален обем:

fdisk -BI md0

5) Записва се стандартен label и bootstrap код:

bsdlabel -w -B md0s1

6) Създават се нужния на брой slices (обикновено само md0s1a):

bsdlabel -e md0s1

7) Създава се файлова система върху новите slices:

newfs /dev/md0s1a (**newfs -b 8192 -f 1024 /dev/md0s1a**)

8) Намалява се заделяното при свръх потребление място:

tunefs -m 0 /dev/md0s1a

9) Монтира се така създадения псевдо диск в работната директория:

mount /dev/md0s1a /core

10) Създава се необходимата структура от директории:

mtree -U -p /core -f /etc/mtree/BSD.root.dist
mtree -U -p /core/usr/ -f /etc/mtree/BSD.usr.dist
mtree -U -p /core/var -f /etc/mtree/BSD.var.dist

11) Използват се помагалата от проекта miniBSD (https://neon1.net/misc/minibsd.html). Тези помагала са избрани защото активно се разработват и са доказали своята функционалност. Те се състоят от един конфигурационен файл (minibsd5.files), съдържащ желаните системни помагала и допълнителни файлове, който ще присъстват в окончателния build, както и два perl скрипта, единият (mkmini.pl), от който, копира помагалата от host операционната система в работната директория (респ. Върху подготвения image), а другият (mklibs.pl) създава списък с необходимите системни библиотеки за правилната работа на системните помагала.

Редактира/Създава се списък с необходимите файлове/помагала:

vi minibsd5.files

Помагалата/Файловете от списъка се копират в работната директория:

perl mkmini.pl minibsd5.files / /core/

Генерира се списък с необходимите библиотеки:

perl mklibs.pl /core/ > minibsd5.libs

Копират се необходимите библиотеки в работната директория:

perl mkmini.pl minibsd5.libs / /core/
cp -p /usr/lib/pam* /core/usr/lib

12) Копира се конфигурационната директория /etc:

cp -R /etc /core/etc

13) Редактира се **/core/etc/rc.conf** според изискванията към embedded build-a:

vi /core/etc/rc.conf

14) Създава се **/core/etc/fstab** със подходящо съдържание:

В случая:

```
/dev/ad0s1a     /       ufs     ro      1       1
proc            /proc   procfs  rw      0       0
/dev/md0        /tmp    mfs     rw,-s=8192      0       0
/dev/md1        /var    mfs     rw,-s=8192      0       0
```

където „-s=8192“ показва големината в 512-байтови на създадените memory disks, където ще бъдат разположени /var и /tmp по време на изпълнение.

15) Създава се файл с потребителски имена и пароли:

```
rm /core/etc/master.passwd
pwd_mkdb -d /core/etc/ /core/etc/master.passwd
```

16) Задава се парола на административния потребител:

```
chroot /core/
passwd root
exit
```

17) Създава се и се копира ядрото на операционната система:

```
cd /usr/src/sys/i386/conf
cp GENERIC mini-kernel
vi mini-kernel
cd /usr/src
make buildkernel KERNCONF=mini-kernel
make installkernel KERNCONF=mini-kernel DESTDIR=/core
```

В случай на недостатъчно място в image е възможно да се пропусне последната стъпка като вместо да се инсталира ядрото и всички възможни модули, администраторът прецени кои от тях са нужни и копира тях и ядрото в /core/boot/kernel.

18) Заключителен етап:

```
umount /core
mdconfig -d -u md0
```

Така подготвения файл drive.img съдържа напълно функционално умалено копие на host операционната система и готов за прехвърляне върху устройството от което ще стартира embedded системата.

Възможно е процесът да се унифицира в определена степен. Така създаденият image файл може да се ползва за създаване и тестване на бъдещият build (примерно чрез използване на емулатори като Bochs или Qemu на локално ниво). В момента, в който build-ът е достатъчно завършен неговото съдържание може да бъде „събрано" с помощта на tar, dump или cpio и прехвърлено върху вече подготвена bootable среда. Така се решават проблеми с разпространението на един и същ build върху носители с различни характеристики или стартиращи среди, различни от стандартната за операционната система.

7.2 **Използване на статично свързани изпълними файлове**

Вторият основен подход се състои в компилиране на специално ядро с ramdisk поддръжка. Следващата стъпка включва създаване на image, в който се намират необходимите системни помагала за правилната работа на системата. Последната стъпка се състои във вграждане на така създаденият image в ядрото. Резултатът е kernel, в който се намира копие на файловата система, която се зарежда в паметта при стартиране и съдържанието и се ползва от там. Предимството на подхода е в това, че се създава само един файл, което в някой случай улеснява процеса на стартиране на операционната система; всички изпълними файлове са статично свързани и интегрирани във файловата система в ядрото.

Изисквания към средата:

- Операционна Система: NetBSD

- Необходими:
    - source кодът на операционната система
    - Bochs / Qemu (за тестове)

Изграждане на embedded BSD build:

1) Подготовка на ядрото:

```
options   MEMORY_DISK_HOOKS
options   MEMORY_DISK_IS_ROOT      # root on ramdisk
options   MEMORY_DISK_SERVER=1     # writeable ramdisk
options   MEMORY_DISK_ROOT_SIZE=8192    # 4Meg
options   MEMORY_RBFLAGS=0         # multi user
```

2) Изграждане на ядрото:

```
config MYTINY
cp ../compile/MYTINY
make depend; make
```

3) Crunched binaries:

Тъй като динамичното свързване не е възможно в разглеждания сценарий, а статичното свързване на библиотеки към всеки един tool наличен в системата би било разхищение на ресурси се ползва интересен трик. Програмата **crunchgen** „взема" обектните файлове на желаните tools и ги свързва в един обектен файл, към който се линкват всички необходими библиотеки. Така се създава само един instance съдържащ едновременно всички необходими системни помагала и изискваните от тях библиотеки. Отговорът на въпроса как се разбира кое точно приложение да се стартира се състои в използването на hard links. Според това под какво име (argv[0]) е било извикано binary-то се стартира и съответното системно помагало.

Примерен конфигурационен файл за crunchgen (**crunch.conf**) би изглеждал по следния начин:

```
srcdirs /usr/src/bin /usr/src/sbin
progs init sh reboot ls
ln sh -sh
libs -lutil -lcrypt -ll -ledit -ltermcap
```

4) Създаване на crunched binary:

```
crunchgen -m Makefile crunch.conf
make -f Makefile objs exe
```

Тези две команди генерират необходимия изпълним файл съдържащ всички описани в конфигурационния файл помагала, както и необходимите за работата им системни библиотеки. Резултатният файл се казва „crunch" (crunch.conf без разширението „conf").

5) Подготовка на image, съдържащ файловата система:

```
mkdir files
mkdir files/bin files/sbin files/dev
```

6) Създаване на необходимите устройства в /dev

```
cp /dev/MAKEDEV files/dev
cd files/dev
./MAKEDEV floppy ramdisk wscons
```

```
cd ../..
```

7) Попълване на файловата система:

```
cp crunch files/sbin/init
ln files/sbin/init files/bin/sh
ln files/sbin/init files/bin/ls
ln files/sbin/init files/sbin/reboot
```

8) Фиксиране на правата:

```
su root
chown -R root:wheel files
exit
```

9) Създаване на image за вграждане в ядрото:

```
makefs -s 4m -t ffs crunch.image files
```

10) Вграждане на image в ядрото:

```
mdsetimage netbsd.ramdisk crunch.image
```

11) Заключителни:

```
gzip -c netbsd.ramdisk > netbsd
```

Така създаденият kernel (netbsd) съдържа ядрото на операционната система и съдържанието на ramdisk-а, от който се стартират системните помагала. Конфигурацията е минимална, но въпреки това функционална. Съдържа само init, ls и reboot. Полученото ядро може да бъде разположено на boot дискета, диск или някаква друга медия, откъдето да бъде зареждано при стартиране на системата.

8. **Next Steps**

Разгледаните до тук последователности от действия създават относително завършени и функционални среди за работа. В голяма част от случаите това е достатъчно. Доколкото целта е embedded системата да може да обслужва процеси, за обслужване, на които вече съществува код в съответната BSD дистрибуция, това е достатъчно.

Съществуват обаче ситуации, при които изграждането на embedded build с операционната система е само стъпка към завършване на цялостния проект.

Разглеждане на софтуерни проекти обслужващи устройства и процеси, работещи в embedded BSD среда е извън обхвата на настоящото изложение.

Все пак от дизайнерска гледна точка е важно да бъде преценено добре къде точно в архитектурата на операционната система е най-добре да бъдат внасяни промените или допълнителният код. От една страна е възможно да се модифицира самият init процес, но от друга е възможно желаният код да се организира като отделно приложение, за да не се губи гъвкавостта от използване на rc системата и на на системния shell. Промяната на init процеса би улеснила вкарването на код за изпълнение след стартиране, но от друга страна завършената многопотребителска среда, която нормално съществува в BSD предоставя контрол на достъпа до системните ресурси.

Изводът е, че към всеки embedded проект трябва да се подхожда

индивидуално и да се търси най-удачният подход за организиране, както на операционната система, така и на заобикалящата я среда и приложения.

Приложение 1: **Links**

- **Дистрибуции:**

http://www.freebsd.org/
http://www.netbsd.org/
http://www.openbsd.org/
http://www.dragonflybsd.org/

- **Embedded Дистрибуции и помагала за изграждането им:**

https://neon1.net/misc/minibsd.html
http://www.ultradesic.com/index.php?section=86
http://www.tinybsd.org/tinybsd
http://people.freebsd.org/~phk/nanobsd/
http://www.wifibsd.org/
http://www.fml.org/software/fdgw/
http://people.freebsd.org/~picobsd/
http://thewall.sourceforge.net/
http://www.closedbsd.org/
http://www.floppy-1.com/
http://www.3bit.co.jp/shapeip/
http://m0n0.ch/wall/
http://mobsd.sourceforge.net/
http://www.theapt.org/openbsd/firewall.html
http://www.nmedia.net/~chris/soekris/
http://www.feu-nrmf.ph/norbert/projects/psygnat/

- **Articles:**

http://www.bsdnewsletter.com/2003/09/Features102.html
http://www.onlamp.com/pub/a/bsd/2004/03/11/Big_Scary_Daemons.html
http://www.teamten.com/lawrence/291.paper/291.paper.html

- **Tools:**

http://people.freebsd.org/~picobsd/tinyware/
http://www.fefe.de/dietlibc/
http://www.uclibc.org/