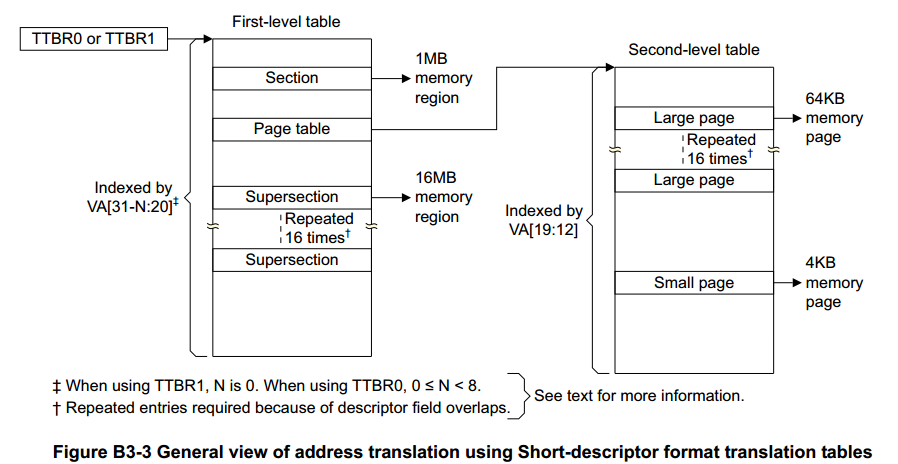
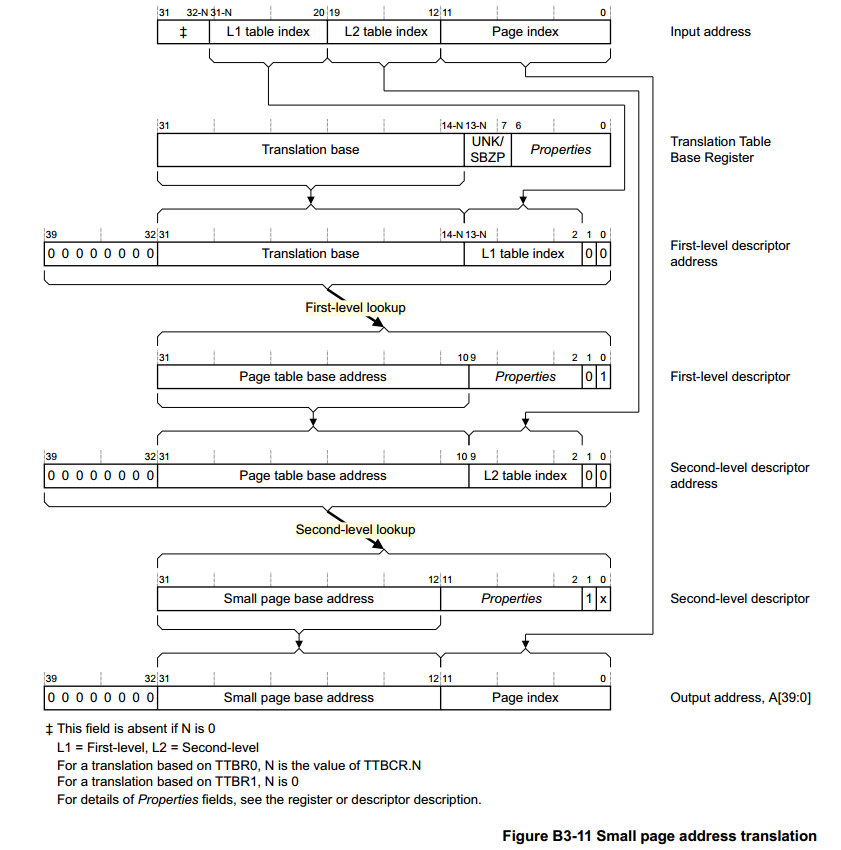
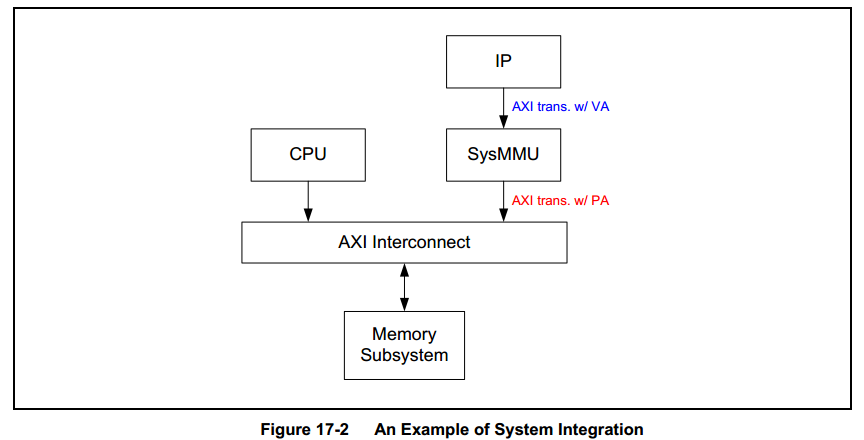
ARM linux 内存管理-页表机制

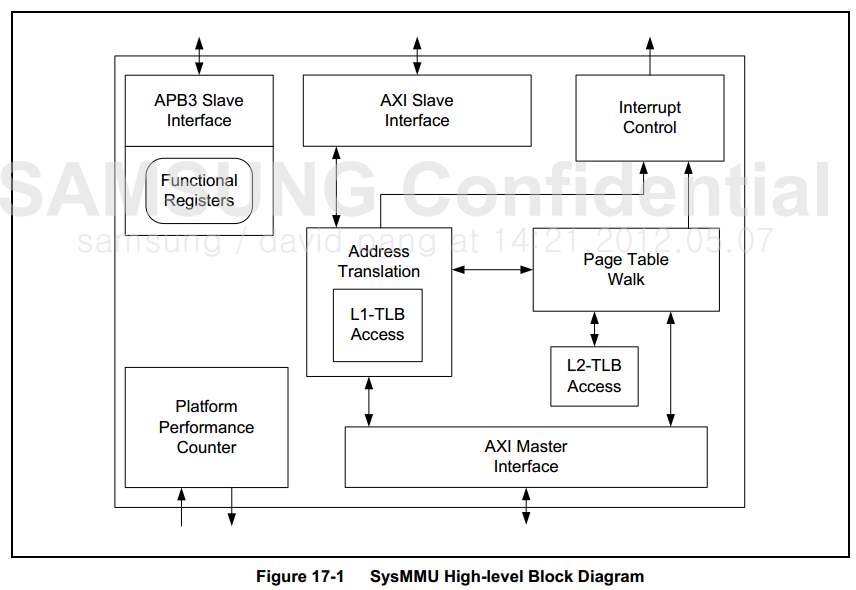
1 ARM硬件上对内存映射的支持

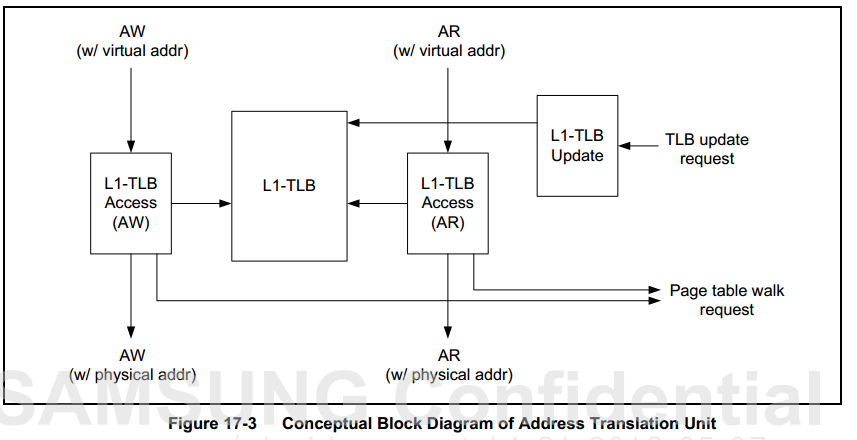


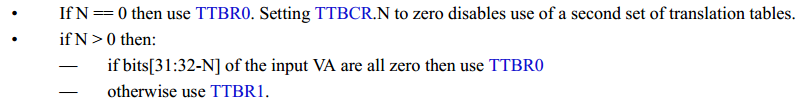


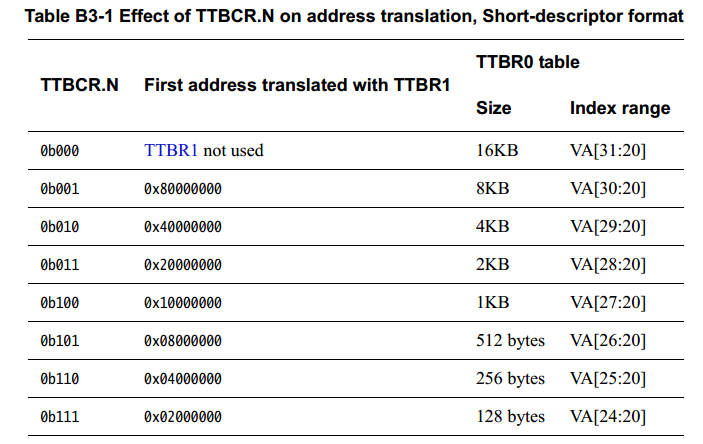
Exynos4412:



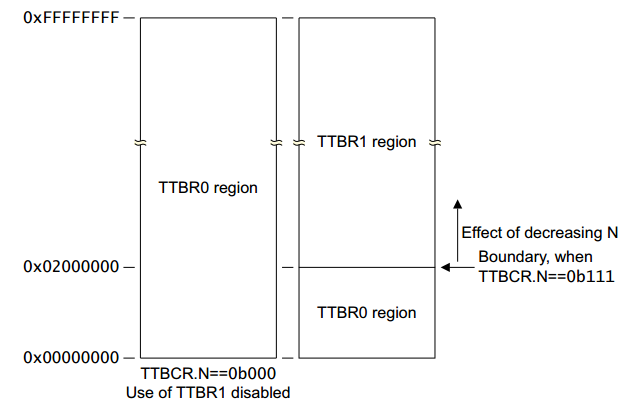








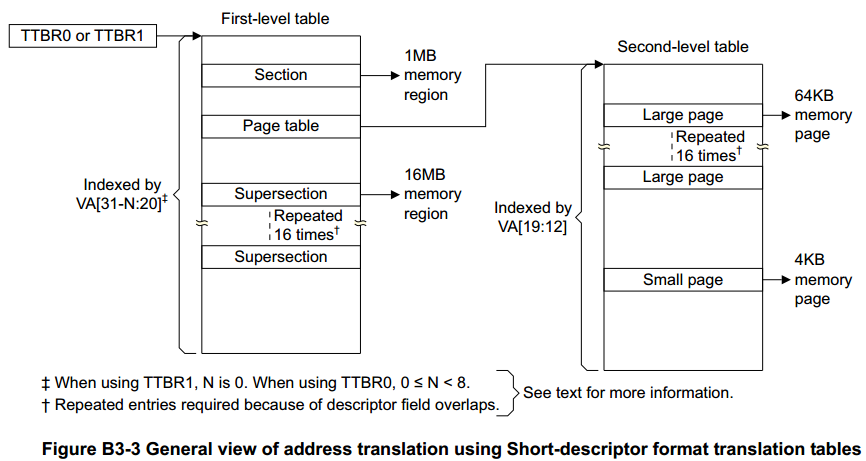
当TTBCR.N为1时：0x80000000~以上的VA地址使用TTBR1指向的地址映射表来翻译，而在0x0000000~0x80000000之间的VA则使用TTBR0指向的地址映射表来翻译。



ARMv7规定了2种格式的地址映射表：

1. short descriptor format：针对32bit系统。
2. Long descriptor format：针对64bit系统。

这里我们主要基于short descriptor format来说明地址转换。基于short descriptor format的虚拟地址转换过程如下表：

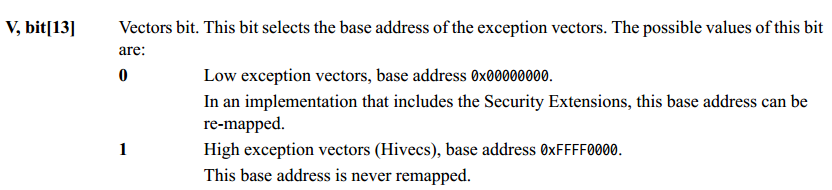


从上图我们看出地址转换分为First-level table 和 Second-level table两个过程，以及映射表是必须的，二级映射表是可选的。

translation table walk :页表查询

每个用户进程在创建的时候会预先分配好16K的页目录表，即每个进程都有自己的页目录表。用户进程的一级页表在创建的时候会拷贝内核的一级页表entry。所有的进程都共用这一份内核页表entry。

除了地址映射表的配置，arm还需要配置异常向量表，如出现data abort,prefetch abort等就需要跳转到对应的异常向量表入口，然后执行预设的异常处理函数。系统启动后默认的异常向量表位置为0x0000000,通过配置CP15的**SCTLR.V** 寄存器的第13位可以指定exception vector需要放置的位置，即低地址0x00000000,或高地址0xffff0000。基本上基于arm的操作系统都是将**SCTLR.V配**置为1。系统出现异常后从此地址开始访问异常向量表。



既然OS指定将异常向量表放到0xffff0000的位置，那么就需要OS将预先设定好的异常向量表拷贝到此位置：

void \_\_init early\_trap\_init(void \*vectors\_base)

{

#ifndef CONFIG\_CPU\_V7M

unsigned long vectors = (unsigned long)vectors\_base;

extern char \_\_stubs\_start[], \_\_stubs\_end[];

extern char \_\_vectors\_start[], \_\_vectors\_end[];

unsigned i;

vectors\_page = vectors\_base;

/\*

\* Poison the vectors page with an undefined instruction. This

\* instruction is chosen to be undefined for both ARM and Thumb

\* ISAs. The Thumb version is an undefined instruction with a

\* branch back to the undefined instruction.

\*/

for (i = 0; i < PAGE\_SIZE / sizeof(u32); i++)

((u32 \*)vectors\_base)[i] = 0xe7fddef1;

/\*

\* Copy the vectors, stubs and kuser helpers (in entry-armv.S)

\* into the vector page, mapped at 0xffff0000, and ensure these

\* are visible to the instruction stream.

\*/

memcpy((void \*)vectors, \_\_vectors\_start, \_\_vectors\_end - \_\_vectors\_start);

memcpy((void \*)vectors + 0x1000, \_\_stubs\_start, \_\_stubs\_end - \_\_stubs\_start);

kuser\_init(vectors\_base);

flush\_icache\_range(vectors, vectors + PAGE\_SIZE \* 2);

#else /\* ifndef CONFIG\_CPU\_V7M \*/

/\*

\* on V7-M there is no need to copy the vector table to a dedicated

\* memory area. The address is configurable and so a table in the kernel

\* image can be used.

\*/

#endif

}

问题：

1. 内核是如何布局内核空间的？
2. 内核是如何组织物理内存的？

ARM linux 内存管理-内存分配

第3章 linux内核调试

要深刻的学习和理解内核的各种机制，就必须要read the fuck source code！然而有些内核代码晦涩难懂，运行过程结果也是千变万化，迷雾重重，有时候结果并不一定是我们想象中的结果，为了准确的把握内核的运行脉搏，我们就需要通过一定的手段对内核进行望闻问切。而最好的方式方法就是使用内核调试工具。同时，为了做到产品性能稳定可靠，对于在内核以及驱动开发过程中，熟练掌握相关的调试手段无疑为我们快速定位和分析问题提供了强大的必杀绝招。诸如gdb，kgdb，ftrace，systrace，crashdump等。本章就来修炼下涉及内核调试的那些必杀技能。

所谓工欲善其事必先利其器，要调试内核我们必须要有一个arm linux内核开发环境。但对于我等新时代的房奴屌丝新手小白来说，入手一块几百大洋到开发板那简直是要我们到命啊。好在开源让我们又一次节省了白花花到银子，使用QEMU开源仿真器就可以在其上仿真模拟一块ARM开发板以及相关的一些外设器件。我们后续调试环境都将基于Ubuntu16.04+QEMU2.8+linux4.9 ARM vexpress环境。

3.1 搭建QEMU模拟 ARM vexpress运行环境

3.1.1 环境介绍

win7 64 + virtualbox5.1.4+ubuntu14.04

qemu：stable qemu-2.8.0

linux kernel：longterm 4.9.34

busybox : busybox-1.24.0

cross compile : arm-none-linux-gnueabi-gcc-4.8.3

3.1.2 工具下载交叉编译

1. 源码准备：

* linux kernel-4.9.34：

$ wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.9.34.tar.gz

* qemu-2.8.0：

$ wget http://download.qemu-project.org/qemu-2.8.0.tar.xz

* busybox-1.24.0

$ wget <https://busybox.net/downloads/busybox-1.24.0.tar.bz2>

* cross-compile/rootfs文件下载:

$ wget https://github.com/gavin-cdwang/qemu-debug-kernel/raw/master/tools/arm-crosstool.tar.bz2

* mkimage，etc文件下载

$ wget <https://github.com/gavin-cdwang/qemu-debug-kernel/raw/master/tools/script-tools.tar>

2. 交叉编译源码

* 相关库安装

sudo apt-get install libcap-dev zlib1g-dev libglib2.0-0 libglib2.0-dev libtool libsdl1.2-dev autoconf flex

* 编译qemu

$ tar -jxvf qemu-2.8.0.tar.xz

$ cd qemu-2.8.0

$ mkdir build/

$ cd build/

$ ../configure --target-list=arm-softmmu,i386-softmmu,x86\_64-softmmu,aarch64-linux-user,arm-linux-user,i386-linux-user,x86\_64-linux-user,aarch64-softmmu --audio-drv-list=alsa

$ make -j8

$ sudo make install

* 编译arm vexpress内核镜像

$ tar -zxvf linux-4.9.34.tar.gz

$ cd linux-4.9.34

$ export CROSS\_COMPILE=/opt/toolchains/arm-2014.05/bin/arm-none-linux-gnueabi-

$ export ARCH=arm

$ make vexpress\_defconfig

$ make menuconfig

配置用户和内核空间为 3G/1G以及高端内存

Kernel Features

Memory split (3G/1G user/kernel split) --->

[\*] High Memory Support

配置ramdisk 8M支持，以便后续好挂载ramdisk文件系统

Block devices

<\*> RAM block device support

(16) Default number of RAM disks (NEW)

(8192) Default RAM disk size (kbytes)

保存后退出，编译内核

$ make -j8

分别生成所需要zImage和dtb：

arch/arm/boot/zImage

arch/arm/boot/dts/vexpress-v2p-ca9.dtb

* 制作ramdisk 根文件系统

首先利用busybox编译基础文件

$ tar -jxvf busybox-1.24.0.tar.bz2

$ cd busybox-1.24.0

$ export CROSS\_COMPILE=/opt/toolchains/arm-2014.05/bin/arm-none-linux-gnueabi-

$ export ARCH=arm

$ make menuconfig

将busybox配置为静态编译

Busybox Settings

Build Options--->

[\*] Build BusyBox as a static binary (no shared libs)

保存退出编译

$ make install

编译完成后在当前目录下生成如下文件：

$ ls -l \_install/

总用量 12

drwxrwxr-x 2 gavin gavin 4096 6月 29 16:21 bin

lrwxrwxrwx 1 gavin gavin 11 6月 29 16:21 linuxrc -> bin/busybox

drwxrwxr-x 2 gavin gavin 4096 6月 29 16:21 sbin

drwxrwxr-x 4 gavin gavin 4096 6月 28 14:02 usr

上述只是生成了相关的命令工具集，对于一个文件系统来说还需要增加相关的设备文件如/dev/console便于printk串口打印，增加配置文件，以便挂载内核动态创建的虚拟文件系统如 sysfs，debugfs，proc等。

制作ramdisk文件系统：

$ mkdir rootfs

$ cp -rfa busybox-1.24.0/\_install/\* rootfs/

$ cp -rfa busybox-1.24.0/examples/bootfloppy/etc/ rootfs/

$ cp -rfa examples/bootfloppy/etc/ \_install/

$ mkdir -p rootfs/proc

$ mkdir -p rootfs/sys

$ mkdir -p rootfs/tmp

$ mkdir -p rootfs/var

$ mkdir -p rootfs/mnt

拷贝c动态库文件，方便后续的应用程序依赖库文件：

$ cp -rfa /opt/toolchains/arm-2014.05/arm-none-linux-gnueabi/libc/lib/ rootfs/

去除符号表减小ramdisk体积：

$ arm-none-linux-gnueabi-strip rootfs/lib/\*

创建dev设备文件，可以看下我们ubuntu系统的样子：

$ ls -l /dev/console

crw------- 1 root root 5, 1 6月 12 17:19 /dev/console

$ ls -l /dev/null

crw-rw-rw- 1 root root 1, 3 6月 12 17:19 /dev/null

$ mkdir -p rootfs/dev

$ sudo mknod rootfs/dev/console c 5 1

$ sudo mknod rootfs/dev/null c 1 3

修改相关配置文件

修改rootfs/etc/inittab文件：

::sysinit:/etc/init.d/rcS

::respawn:-/bin/sh

::askfirst:-/bin/sh

::ctrlaltdel:/bin/umount -a -r

::restart:/sbin/init

修改rootfs/etc/init.d/rcS文件：

#! /bin/sh

/bin/mount -a

mkdir -p /dev/pts

mount -t devpts devpts /dev/pts

echo /sbin/mdev > /proc/sys/kernel/hotplug

mdev -s

ifconfig lo 127.0.0.1

ifconfig eth0 192.168.199.136

/bin/hostname -F /etc/hostname

修改fstab文件内容，以便挂载虚拟文件系统

proc /proc proc defaults 0 0

sysfs /sys sysfs defaults 0 0

debugfs /sys/kernel/debug debugfs defaults 0 0

tmpfs /tmp tmpfs defaults 0 0

tmpfs /dev tmpfs defaults 0 0

制作ramdisk磁盘镜像：

$ sudo dd if=/dev/zero of=ramdisk bs=1M count=8

$ sudo mkfs.ext4 -F ramdisk

$ mkdir tmp

$ sudo mount -t ext4 ramdisk tmp/

$ sudo cp -rfa rootfs/\* tmp/

$ sudo umount tmp/

$ gzip -9 ramdisk

$ ./mkimage -A arm -O linux -T ramdisk -C gzip -n "ramdisk" -d ramdisk.gz ramdisk.img

至此我们成功的制作出了ramdisk文件系统镜像。

为了方便，将上述操作做成一个脚本：

#!/bin/bash

WORK\_DIR=$(pwd)

BUSYBOX\_DIR=$WORK\_DIR/busybox-1.24.0

CROSS\_COMPILE=/opt/toolchains/arm-2014.05

sudo rm -rf $WORK\_DIR/rootfs

sudo rm -rf $WORK\_DIR/tmp

sudo rm -rf $WORK\_DIR/ramdisk\*

sudo mkdir $WORK\_DIR/rootfs

ROOTFS\_DIR=$WORK\_DIR/rootfs

sudo cp -rfa $BUSYBOX\_DIR/\_install/\* $ROOTFS\_DIR

sudo cp -rfa $WORK\_DIR/etc/ $ROOTFS\_DIR

sudo mkdir -p $ROOTFS\_DIR/proc/

sudo mkdir -p $ROOTFS\_DIR/sys/

sudo mkdir -p $ROOTFS\_DIR/tmp/

sudo mkdir -p $ROOTFS\_DIR/var/

sudo mkdir -p $ROOTFS\_DIR/mnt/

sudo cp -arf $CROSS\_COMPILE/arm-none-linux-gnueabi/libc/lib $ROOTFS\_DIR

sudo $CROSS\_COMPILE/bin/arm-none-linux-gnueabi-strip $ROOTFS\_DIR/lib/\* 2>/dev/null

sudo mkdir -p $ROOTFS\_DIR/dev/

sudo mknod $ROOTFS\_DIR/dev/tty1 c 4 1

sudo mknod $ROOTFS\_DIR/dev/tty2 c 4 2

sudo mknod $ROOTFS\_DIR/dev/tty3 c 4 3

sudo mknod $ROOTFS\_DIR/dev/tty4 c 4 4

sudo mknod $ROOTFS\_DIR/dev/console c 5 1

sudo mknod $ROOTFS\_DIR/dev/null c 1 3

echo

echo "@@@@@ make the ramdisk and format to ext4 @@@@@"

sudo dd if=/dev/zero of=ramdisk bs=1M count=8

sudo mkfs.ext4 -F ramdisk

sudo mkdir -p tmp

sudo mount -t ext4 ramdisk tmp/

sudo cp -raf $ROOTFS\_DIR/\* tmp/

sudo umount tmp

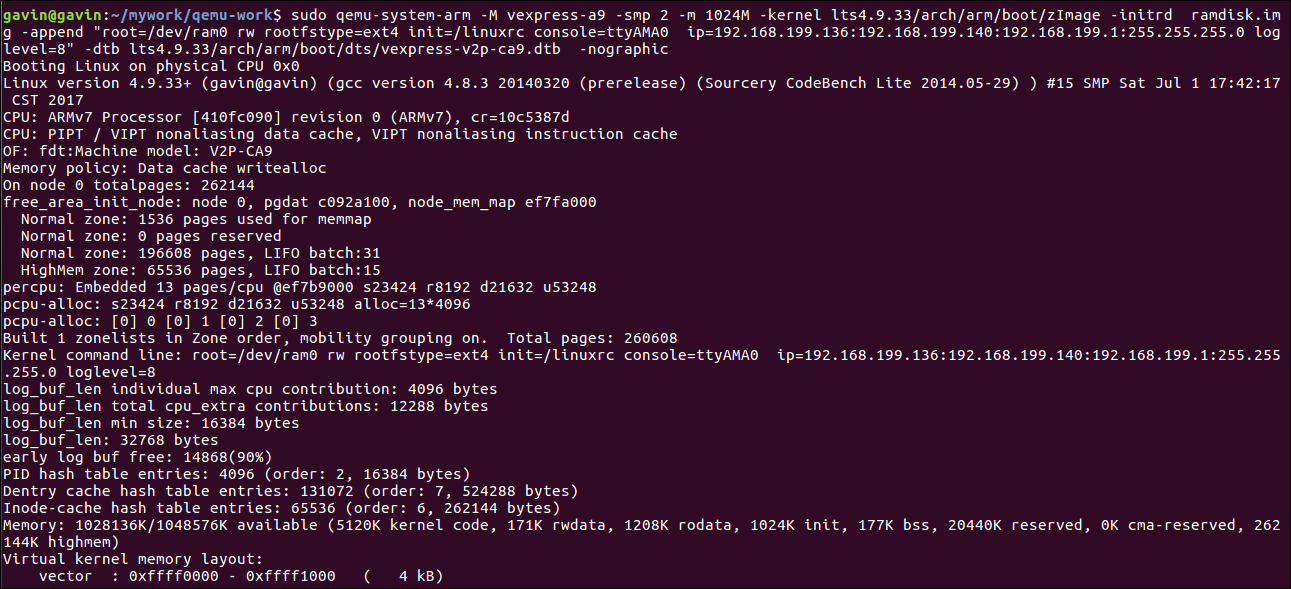
gzip -9 ramdisk

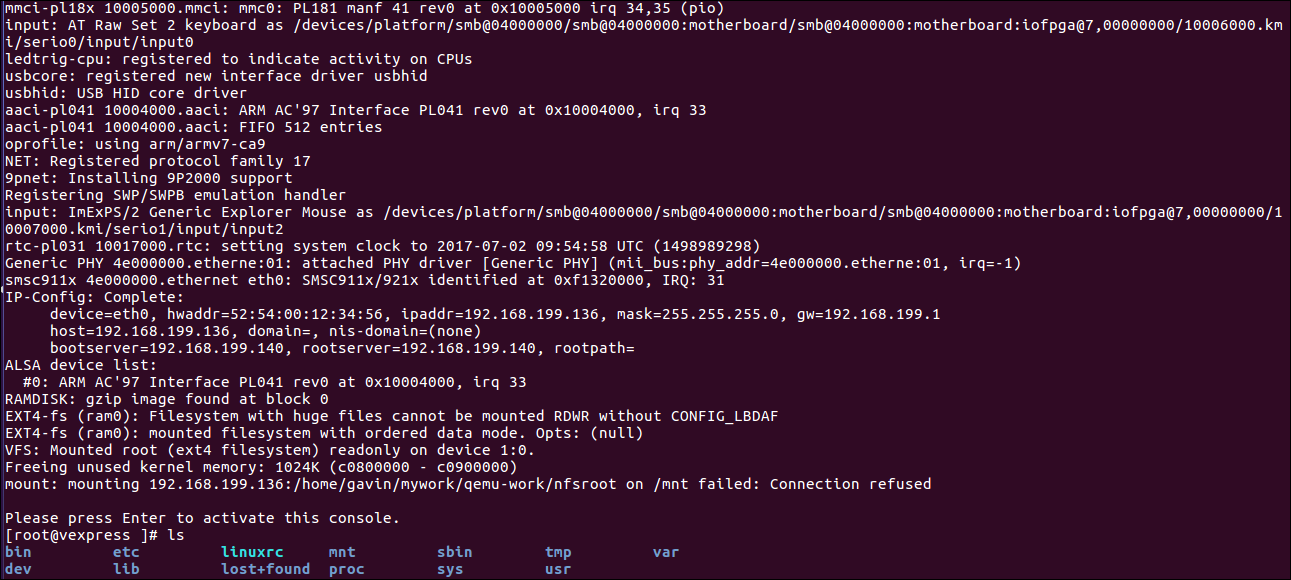
echo "@@@@@make the ramdisk to fit uboot format @@@@@"

sudo mkimage -A arm -O linux -T ramdisk -C gzip -n "ramdisk" -d ramdisk.gz ramdisk.img

3.1.3 启动qemu运行

qemu-system-arm -M vexpress-a9 -smp 2 -m 1024M -kernel ./lts4.9.33/arch/arm/boot/zImage -initrd ramdisk.img -append "root=/dev/ram0 rw rootfstype=ext4 init=/linuxrc console=ttyAMA0 ip=192.168.199.110:192.168.199.130:192.168.199.1:255.255.255.0 loglevel=8" -dtb ./lts4.9.33/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic





3.1.4 支持网络挂载nfs

由于ramdisk是只读的，为了方便后续的调试，文件修改，我们让qemu支持网络挂载nfs。一般我们使用qemu通过TAP/TUN接口以桥接的方式创建虚拟网卡来支持网络。

这需要ubuntu主机（即内核）系统支持TUN/TAP，我们现在使用ubuntu基本都是支持的，可以通过查看一下节点看是否支持：



主要思路：将qemu注册到主机的一个单独网络接口上，然后将这个接口和主机对外的网络接口一起桥接起来。

实际操作：在宿主机中为qemu创建一个tap虚拟网络接口，并创建一个虚拟网桥br0,并将tap和br0，br0和主机对外的网络接口桥接起来就可以实现虚拟机的网络了。

1. 宿主机软件的安装配置

* 安装必要的工具：

$sudo apt-get install bridge-utils uml-utilities

* 配置网桥，虚拟网卡：

脚本qemu-ifup，使用静态配置

#!/bin/sh

# script to bring up the tun device in QEMU in bridged mode

ETH0IPADDR=192.168.199.100

MASK=255.255.255.0

GATEWAY=192.168.199.1

BROADCAST=192.168.199.255

#create the tap0 interface

tunctl -u $USER -t $1

sudo ifconfig enp0s3 0.0.0.0 promisc up

sudo ifconfig $1 0.0.0.0 promisc up

#

# create the bridge between eth0 and the tap device

#

sudo brctl addbr br0

sudo brctl addif br0 enp0s3

sudo brctl addif br0 $1

brctl stp br0 off

#use static config ip

ifconfig br0 $ETH0IPADDR netmask $MASK broadcast $BROADCAST

route add default gw $GATEWAY

sudo ifconfig br0 192.168.199.100 up

sudo ifconfig enp0s3 192.168.199.136 up

sudo ifconfig $1 192.168.199.120 up

1. 支持NFS

* 工具安装：

sudo apt-get install nfs-kernel-server

sudo apt-get install rpcbind

* 编辑exports导出目录：

sudo vim /etc/exports

/home/gavin/mywork/qemu-work/nfsroot \*(rw,sync,no\_root\_squash,no\_subtree\_check)

* 启动服务：

sudo /etc/init.d/rpcbind restart

/etc/init.d/nfs-kernel-server restart

1. 启动qemu，挂载nfs

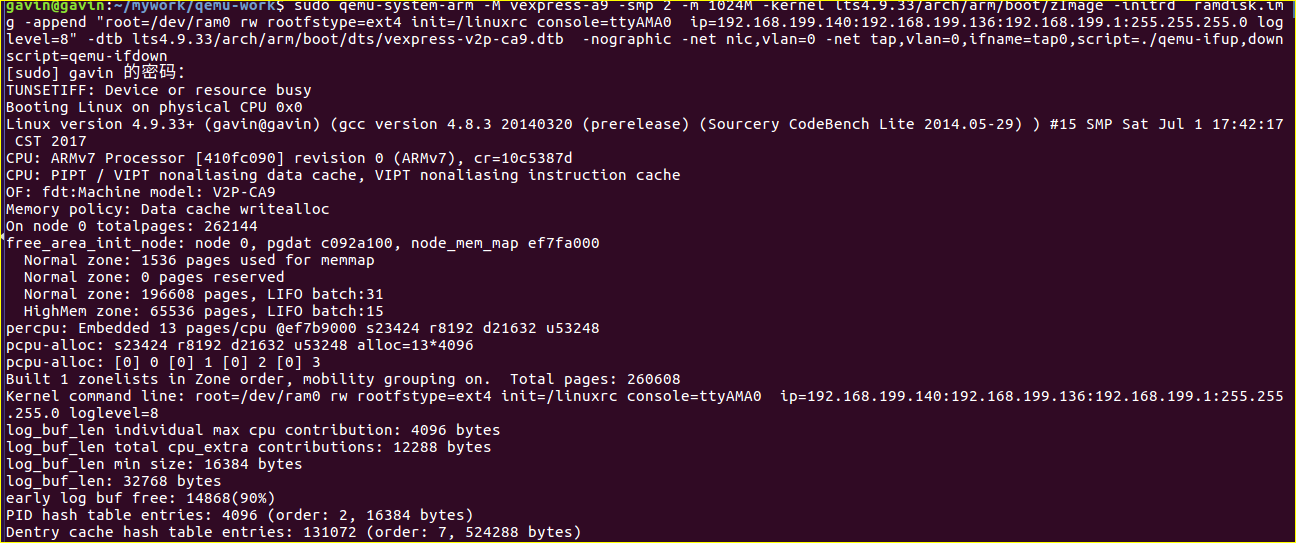
* 在ramdisk的etc/init.d/rcS中挂载nfs：

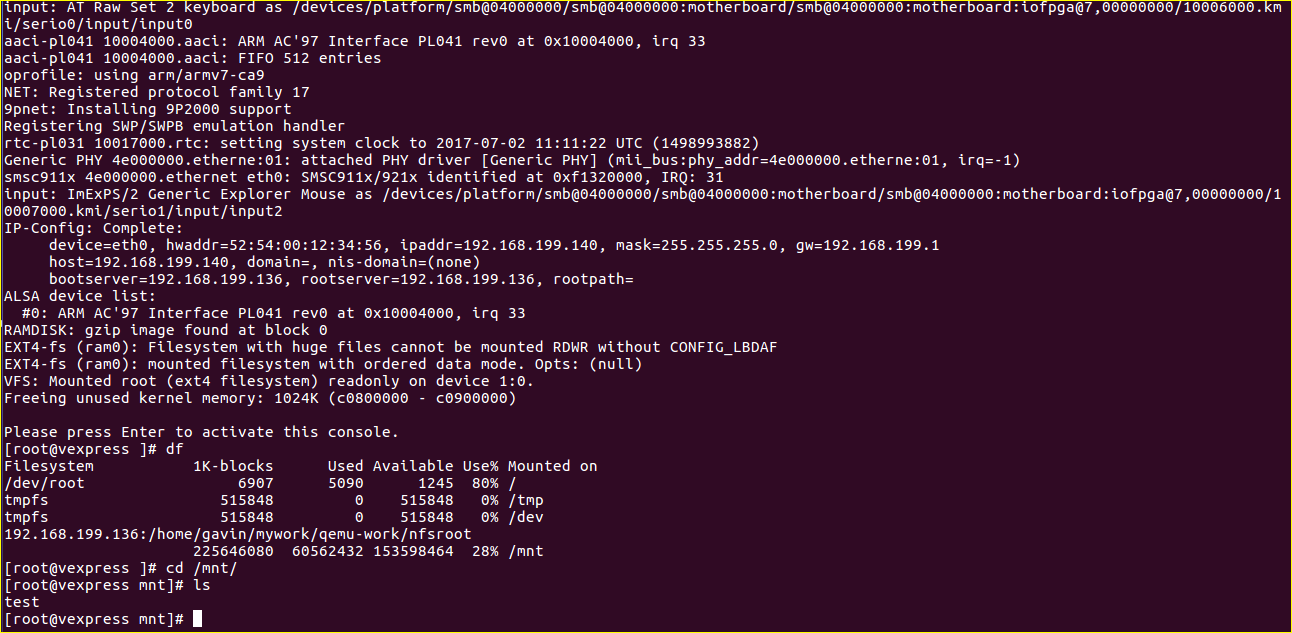
mount -t nfs -o nolock 192.168.199.136:/home/gavin/mywork/qemu-work/nfsroot /mnt

* 启动qemu

launch\_qemu.sh：

sudo qemu-system-arm -M vexpress-a9 -smp 2 -m 1024M -kernel lts4.9.33/arch/arm/boot/zImage -initrd ramdisk.img -append "root=/dev/ram0 rw rootfstype=ext4 init=/linuxrc console=ttyAMA0 ip=192.168.199.140:192.168.199.136:192.168.199.1:255.255.255.0 loglevel=8" -dtb lts4.9.33/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic -net nic,vlan=0 -net tap,vlan=0,ifname=tap0,script=./qemu-ifup,downscript=qemu-ifdown





可以看到已经成功挂载了nfsroot目录访问到了test。

注:上述的工具脚本mkimage，qemu-ifup，qemu-ifdown，launch\_qemu.sh mk\_ramdisk.sh我已经上传到github。

gavin@work:~/mywork/project/avl7420\_porting/qemuwork/mkmyroot$ sudo dd if=/dev/zero of=/sddata.img bs=1M count=1024

记录了1024+0 的读入

记录了1024+0 的写出

1073741824字节(1.1 GB)已复制，2.14214 秒，501 MB/秒

git 操作

添加sshkey后执行：

$ ssh -T [git@github.com](mailto:git@github.com)



建立仓库：

git init

git add ./

git commit -m “xxxxxx”

本地仓库关联远程github仓库

git remote add origin <https://github.com/gavin-cdwang/qemu-debug-kernel.git>

上传github前，先要pull下：

git pull origin master

gavin@work:~/mywork/project/avl7420\_porting/qemuwork$ git pull origin master

warning: no common commits

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (7/7), done.

remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (7/7), done.

来自 https://github.com/gavin-cdwang/qemu-debug-kernel

\* branch master -> FETCH\_HEAD

\* [新分支] master -> origin/master

Merge made by the 'recursive' strategy.

.gitignore | 52 ++++++++++++++++++++++++++++++++++++++++++++++++++++

README.md | 3 +++

2 files changed, 55 insertions(+)

create mode 100644 .gitignore

create mode 100644 README.md

gavin@work:~/mywork/project/avl7420\_porting/qemuwork$

上传代码到github远程仓库

git push -u origin master