清华大学计算机科学与技术系
计算机组成原理挑战课暨软件工程联合大实验

操作系统课程设计最终文档

--Serial, Flash and Debugger

清华大学计算机系
李宇轩

组员 李宇轩 2013011349
组员 董豪宇 2013011356
组员 梁泽宇 2013011346

目录

1 引言

1.1 项目背景

本次实验，为挑战性课程的操统部分实验，是上个学期软件工程、计算机组成原理课程联合项目的后续。

承担本项目的开发者为计 33 班的李宇轩，董豪宇和梁泽宇同学。

和上学期一样，本次实验是一个软硬件协同开发的工作，因此要求开发者既能够理解 CPU 的底层实现，也能够了解在此基础上的操作系统的行为。不同的是，由于是进行功能扩展，三个人分别进行不同的扩展，工作相对于较为独立。

具体分工为，董豪宇同学负责文件系统的修改，梁泽宇同学负责网络模块的支持，李宇轩同学则负责 Debugger 模块和处理一些硬件问题。

本文将总结我——李宇轩部分的工作——串口，Flash 和 Debugger 模块。

1.2 编写目的

编写此文档的目的主要在于总结现阶段工作，并且为之后的学弟提供相应地支持与便利。

1.3 文档结构

先从较为简单的串口入手，再解释 Flash 的相关工作，最后则是工作的重心 Debugger 模块。

1.4 我的分工

我在本次课程设计中，完成的工作为较为独立地实现串口、Flash 和 Debugger 模块。具体的工作本文接下来的篇幅将会具体讲解。

2 串口模块

2.1 基本介绍

计算机内部数据通常以字或字节为单位进行并行访问。但与外部设备进行数据交互或与其他计算机进行通信时，也经常需要串行数据通信。串行接口是一类使用相对简单的接口，thinpad 教学计算机上也配置了基本的串口，作为连接中断设备的接口。

2.2 实验需求

ucore 和所有的操作系统一样，可以提供最基本的 I/O 支持。实现这个 I/O 我们有很多的方法。比如说通过 VGA 显示，通过 PS2 键盘输入。但是有更简单的方式，就是直接通过串口通信来同时支持操统的数据输入和输出数据的显示。其中数据的输入就是操统接受到串口传递过来的数据，而数据的显示则是将需要显示的字符发送给远端的 PC 计算机，在 PC 计算机上的串口通信终端上显示。

上学期已将基本的串口实现，但是通过测试发现，使用上学期的串口传文件有着严重丢包情况产生，这就使得我们不得不得学习一些跨时钟域通信的协议，写一个鲁棒的串口通信模块。

2.3 实现原理

采样部分可以使用 www.fpga4fun.org 的现成代码，具体原理大概是取多个采样点的中点，一般来说中点的信号比较稳定。

我们假设 cpu 处于一个快时钟域，串口处于一个慢时钟域（分别为 25M 和 11M）。快时钟域向慢时钟域通信的时候，可以使用脉冲同步器；慢时钟域向快时钟域同步，可以使用边沿同步器。

此外，还需要走两层触发器，防止亚稳态的产生。

只要注意了这些细节，实现一个能在 115200 波特率下工作的串口模块，应当是没有问题的。

2.4 测试方案

由于寒假就研究了串口丢包的相关问题，所以串口的实现相对来说是比较简单的。关键工程量就在于测试。

这里采用的测试方法是，pc 端反复发送"12345678" 字符串，在写好的串口模块的基础上搭一个 test_bench，test_bench 每次将收到的字符发送回 pc 端。如果看到 12345678 丢失了几个字符导致了 pc 端终端输出不对弃，则认为串口没有稳定工作。否则则认为串口已经稳定。

之所以这样考虑的原因，是之前观察了串口出错的情况，可以发现肯定是由于丢包而不是误传。在这个假定的基础上，我们认为这样一个检验是基本合理的，如果需要进一步检查，可以把收到的数据使用工具比对。

具体的 test_bench 代码见 git 工程下 work_JFLFY/vhdl_src/test_com。

2.5 遇到的问题

调试串口的过程中未遇到明显的问题。

2.6 实现成果

我们实现的串口通过了设计的测试方案，能在这个 test_bench 下稳定传输至少 1 小时。此外，通过回环测试我测试出了当时使用串口自发自收的最高波特率，为 430400，并且在这个速率下串口也能完全正常工作。

到这里为止，认为串口实验的结果非常理想，满足要求。

3 Flash 模块

3.1 基本介绍

Flash 是存储芯片的一种，通过特定的程序可以修改其中存储的数据。Flash 存储器又称为闪存，它结合了 ROM 和 RAM 的长处，不禁具备电子可擦除和可编程的功能，还能够快速读取数据，Flash 不像普通的闪存，保存的数据断电后也不会丢失。

thinpad 中的 Flash 型号为 MT28F640J3，一共 8MB 的空间，支持 16 位数据的存取操作。

3.2 实验需求

先看看上个学期计软联合大实验的要求：

> thinpad 的 Flash 被定位为一个很简单的外存。这个外存提供操作系统的断电存储。所以事实上，仅仅需要支持 Flash 的读取操作即可，无需支持其写入操作。ucore 操统没有在 Flash 上支持文件的写入。

这学期，我们需要在将 sfs 文件系统进行扩展，使其支持新增、删除文件。由于这个文件系统需要被实现在 Flash 上，我们的 CPU 需要完整的支持对于 Flash 的 erase 和 write。

3.3 实现原理

关于 MT28F640J3 Flash，可以参考的资料主要有上学期的《计算机硬件实验原理》和该芯片的 datasheet。我分别阅读了这两个资料，发现《计算机硬件实验原理》上面的描述虽然是正确的，但是不够完备、准确。

具体来说，flash 具有的基本特性是，只能把一位上的 1 写成 0，而不能把 0 写成 1。所以为了实现往 Flash 存入信息，需要同时实现 Flash 的 erase 和 write 操作。

对于 Flash 的操作满足这样一个模式，先往地址里面写入一个操作码，之后紧跟着的指令就可以执行相应的 read, erase 和 write 操作。他们三者的状态机如下：

- read 的状态机：向 Flash 的任意地址中先写入一个 0xff，表示转换到读 Flash 模式。
- write 的状态机：向要写入 Flash 的对应地址中写入一个 0x40，表示转换到写“字”模式；再往该地址写入需要写入的值。这时候 Flash 会将读模式切换为读状态寄存器模式，从任意地址中读出状态寄存器的值，直到第八位为 1 表示写入完毕，方可进行下一个操作。
- erase 的状态机：向要写入 Flash 的对应的块地址中写入一个 0x20，表示转换到写“字”模式；再往该块地址中写入 0xD0 表示 erase。[1]这时候 Flash 会将读模式切换为读状态寄存器模式，从任意地址中读出状态寄存器的值，直到第八位为 1 表示写入完毕，方可进行下一个操作。

知道了前面的那些协议知识，就可以着手实现 Flash 通讯模块了。我实现 Flash 的时候较早，当时文件系统的工作还没有步入正轨，还不知道需要将 Flash 的接口设计成什么样子。所以我实现了两种不同层次 Flash 接口，一个接口对外暴露了 Flash 先指令后操作的特性，完全通过前面描述的状态机来操作 Flash，需要操统有简单的 Flash 驱动，才能使用这个接口；另一个接口则封装了那个简单的接口，使用这个接口，ucore 可以像访问 ram 一样访问 flash，只是 erase 需要 hack 一条特殊的指令。

当然，最终被选择的是前者，这个模块更为灵活，用软件驱动操作 Flash 模块更灵活，比如可以支持队友董豪宇实现异步擦除操作。

[1] 实测不需要是块开头的地址，写的地址属于哪个块，Flash 就会清空哪个块

3.4 测试方案

3.4.1 测试方案 A

我对 Flash 进行的第一个测试，是通过前面描述的第二个 Flash 接口实现的。可以说这也是当时为何实现这样一个接口的原因，因为有了这样一个可以简单使用的接口，就可以使用以前给 ram 搭的 test_bench 直接进行对于 Flash 的测试。

测试是逐步进行的。先将 clk 设置成手动触发，检测 erase 功能，用 test_bench 清空 Flash 的一个块，在用其他的工具观察这个块的值。通过了第一步以后，再检测 write 功能，往 Flash 中反复写入“12345678”，再用其他的工具观察。经观察无误后，最后测试 Flash 的读取，依次读取 Flash，判断读出的内容是否为“12345678”。

最后取消手动 clk 模式，尝试 Flash 模块能在什么样的时钟下运行，和 datasheet 进行比对，查看实现上面是否有能导致需降频的瑕疵。

这个测试简单而有效，经过测试的 Flash 是能正常工作。

3.4.2 测试方案 B

对于 Flash 的第二次测试是在将已经写好的 Flash 模块放入 CPU 时进行的。由于此时我的 Debugger 模块已经正常工作，可以稳定地从 ram 中读取大量数据，也可以直接作为 ram 的烧写器。我就将 Flash 模块放入操统后，进行了简单的修改，支持了 Debugger 模块对于 Flash 的访问。

只要 Debugger 可以成为 Flash 的烧写器，并且操统可以正常从 Flash 启动，大致就可以认为 Flash 模块正确。但，在接下来的“遇到的问题”小节我们将看到，这个测试看起来十分完美，但是存在检查不出的问题，从而对我们的课程设计产生了巨大的影响。

3.4.3 测试方案 C

这个测试方案与其表达为测试方案，不如说是一个最终目标之一。即支持一个有新版 sfs 文件系统的 ucore。

我们组的情况是这个 ucore 在 qemu 下已经跑通了，可认为其基本正确，只要将其上板测试即可作为最后的测例。

3.5 遇到的问题

3.5.1 问题一

在测试方案 A 中没有遇到明显的问题。

在测试方案 B 和 C 中，出现了通过 B 却无法通过 C 的情况。这个问题也是导致我们 14 周演示效果不佳的原因。具体来说，就是能够正常的烧写 ucore 进 Flash，暂停代码使用 Debugger 访问 Flash 也完全正常，但一旦运行代码就会问题。

通过使用 Debugger 观察 Flash 的运行结果，可以发现其会在莫名其妙的地方转换读模式。而 Flash 转换读模式的只可能是因为向 Flash 的某个地址写入了操作指令。然而问题在于，这段代码明明不应该有写入 Flash 的过程，却产生了写入 Flash 的效果。

我最初的猜想是 Flash 运行的时钟频率太高，导致操统运行不稳定。但这个 Flash 使能信号出问题，是很难通过常规手段调试的，需要示波器、逻辑分析仪类似的手段才会比较方便。在调试毫无进展的时候，巧合地发现我不小心将一个'w' 写成了'o'，使得在某些情况下，会让读使能更新写使能，这是 Flash 出错的最终原因。不难发现，这个问题将会导致之前的诡异情况出现。比如之所以能通过测试 B 而无法通过测试 C，是因为测试 B 不会交替进行读和写，从而不会误更新写使能，而测试 C 却无法避免会触及这一错误。

3.5.2 问题二

问题二是第十六周最终检测时候第一次遇到的。在 ucore 和 CPU 都没有重新编译的情况下，ucore 在板上运行到的 erase Flash block 0 的时候，突然就卡死。

这个问题当场的解决比较巧合，随便使用 Debugger 运行了几个其他程序之后，这个问题就解决了。通过和张宇翔同学的讨论，我们认为之所以卡死是因为 Flash 在上次断电的时候正在工作，从而锁定了该 block，导致一切操作无效。

非正常情况断电会锁定 block 的这一特性在 datasheet 中是有相关说明的。

3.6 实现成果

最终两版 Flash 代码均能正常工作，前一版在 3.125M 的 CPU 下正常工作，后一版在 25M 的 CPU 下正常工作。可以清空，写入，从 Flash 启动操作系统，并且能使用 Debugger 模块正常地操纵 Flash，使用从 Debugger 模块改装的工具将 ucore 正确的烧入 Flash 中，可以作为一个不错的烧 Flash 工具。

4 Debugger 模块

4.1 基本介绍

调试是开发过程中必不可少的环节，通用的桌面操作系统与嵌入式操作系统在调试环境上存在明显的差别。前者，调试器与被调试的程序往往是运行在同一台机器、相同的操作系统上的两个进程，调试器进程通过操作系统专门提供的调用接口控制、访问被调试进程。后者，又称为远程调试，为了向系统开发人员提供灵活、方便的调试界面，调试器还是运行于通用桌面操作系统的应用程序，被调试的程序则运行于基于特定硬件平台的嵌入式操作系统。

嵌入式带来以下问题：调试器与被调试程序如何通信，被调试程序产生异常如何及时通知调试器，调试器如何控制、访问被调试程序，调试器如何识别有关被调试程序的多任务信息并控制某一特定任务，调试器如何处理某些与目标硬件平台相关的信息（如目标平台的寄存器信息、机器代码的反汇编等）。

一般来说有两种方式，一是通过插桩，在目标操作系统和调试器内分别加入某些功能模块，二者互通信息来进行调试，一般用于调试用户态。二是片上调试，在处理器内部嵌入额外的控制模块，当满足了一定的触发条件时进入某种特殊状态，从而达到 debug 的目的，一般由于调试内核。

本次课程设计我需要实现的 Debugger，可以说就是一个嵌入式调试工具，这里通过片上调试模块的方式来解决调试问题。

4.2 实验需求

之前的一节已经将背景知识介绍的相对清楚了，本次实验的需求就是实现一个可以和 GDB 交互的片上调试模块，能正确地调试 ucore 内核。

4.3 实现原理

4.3.1 RSP 协议初步

一开始接触到 Debugger 这个题目，相对来说是没有头绪的。那么任务之处就是调研自己究竟需要了解些什么。通过在网上搜索 GDB 远端调试，逐步寻找找到了 GDB RSP 协议，全称 GDB Remote Serial Protocol。

经过筛选，可以发现，，这两篇文章可以很好的，较为完整的了解这个所谓的 RSP 协议。

前一篇文章是一篇讲解性的文章，首先通过这篇文章我了解的是知道 RSP 的基本格式：

- 读寄存器组：（“g”），eg：$g#67；reply：+$123456789abcdef0…#xx
- 写寄存器组：（“G”），eg：$G123456789abcdef0…#xx；reply：+$OK#9a
- 单步命令（“s”），eg：$s#73
- 继续命令（“c”），eg：$c#63

可以看出，这是一个一个字符代表一个指令的字符协议。+ 号表示 reply，# 号后面跟着是校验和。通过观察指令和自己对于 GDB 实现的猜测，对于一个相对完备的 GDB 片上调试模块，应该只需要支持单步调试“s”、继续运行“c”、读写寄存器“gG”，读写内存“mM”。比如软断点，就可以通过写内存实现。而例如 b if i==0，这类的指令，则是通过每次到达断点都暂停后判断是否继续运行达成的。

之后这篇文章有些 gdb 交互的例子画成的图，对于理解也是很有帮助的，在这里就不贴出了。

而第二篇文档是 GDB 的官方文档。这个文档作为一个字典也是很棒的，可以查到你通讯过程用到所有指令的含义，以及合法的 reply 方式。

4.3.2 RSP 协议实验

通过阅读文章，了解 RSP 协议的大部分内容，但是对于一个完全陌生的协议，没有一个可以运行的程序供于调试学习的话，可能并不能很快的上手。尤其是虽然能明白交互，但是不太清楚一开始的初始化交互部分。

我选择了自己建立一个中转站，输出 Debug 模式下的 qemu 和 gdb 的交互结果（见 work_JFLFY/net_agent）。通过观察它们的交互，并对照指令含义表，我了解了初始化部分的交互，也进一步对于调试部分产生了印象，之前对于 GDB 实现的猜测，在这里也完全得到了证实。具体一点，能得到的、阅读文档无法知道的信息有：如果你不支持一条指令，你就返回空，GDB 会主动去寻找有代替效果的指令再来询问你的 Debugger。

但之后有了更好的途径去了解这个协议。naive-debugger 是上个学期张宇翔 CPU 组实现的一个简单的 Debugger 的 pc 端，通过阅读 naive 组的代码也是对于 RSP 协议的进一步加强理解。

4.3.3 Debugger 初步

之后就可以开始 Debugger 的实现了。在这个最初的版本里，我参照了 naive 组的实现，使用了 naive 组 pc 端的 server，使用它现有的那套协议和片上调试模块通信。这是一个二进制协议，相比原始的 RSP 协议简单了不少，更加适合硬件实现。

首先需要有一个独立的 Debugger 模块，这个模块维护自己的一个状态机，肩负着通过串口接受指令以及通过串口回复返回值的功能。只要有一定的硬件开发经验，这个模块还是比较好实现的。

通过把寄存器接线接到 Debugger 模块，实现 Debugger 模块对于寄存器的读取。

通过用 Debugger 模块送出的地址覆盖 CPU 给 TLB 模块的地址，可以让 Debugger 控制 ram，从而能实现内存的读取。

为了支持暂停和单步调试，需要设计一个巧妙而鲁棒的暂停规则，这里我当时没有仔细想好，一度浪费了不少时间。

值得注意的是，在这里，调试的时候，将 CPU 流水线暂停而不是刷空（这样其实是不太好的，不能正确支持写寄存器、软中断这些的，原因下节会具体说明）。

4.3.4 功能扩展

第一次扩展的时候，我加入了写寄存器和写内存的功能。

为了实现写寄存器的功能，需要把数据传到通用寄存器堆模块，并且需要传递一个写使能。但事实上，这个功能当时是有问题的。这是由于只暂停流水而不清空流水导致的。

为了实现写内存，只需要传递一个写使能给 MMU 模块，并利用这个使能控制读写。剩下的逻辑读内存的时候基本已经实现。值得一提的是，写内存不会出现上面写寄存器的问题。

第二次扩展的时候，我加入了多个硬断点、disable 断点、GDB 控制的暂停、开关控制启动断点、流水线的清空、软断点和观察点功能。下面分别简单的讲解每个扩展的思路：

1. 多个硬断点：将原来的一个硬断点比较逻辑扩充，比较所有硬断点
2. disable 断点：新增一条协议用于 disable 断点，只要修改一个硬断点在硬件上对应的使能位即可。
3. GDB 控制的暂停：新增一条 stop 协议，用于强行停住正在运行的代码。pc 端的 server 需要新开一个线程，原来的线程接受来自 CPU 的中断信号，新开的线程接受来自 GDB 的中断信号，一旦有一个中断信号就表示程序停止了。
4. 开关控制启动断点：之前的版本里，CPU 一开始不会自己开始运行，而是停在 0xb0000000 这个起始地址，需要来自 GDB 的"c" 才能正常运行。在这里变为用板上的一个开关控制 0xb0000000 断点的使能，可以正常启动，也可以暂停在 0xb0000000。
5. 流水线的清空：将单纯的，需要对于整个 CPU 流水部分的逻辑进行不小的修改。具体的修改因 CPU 实现不同而不同，在此不赘述。
6. 软断点：在有了流水线的清空后，只需要识别 break 指令，并且在暂停规则那里加上一条相关的判断即可。

7. 观察点：和硬断点的实现类似，只是不再是对于指令地址进行暂停，而是对于访存地址进行暂停，两者硬件上的逻辑类似。需要区分 watch, awatch 和 rwatch。

4.4 测试方案

Debugger 的测试相对于其他的模块来说，方法相对简单直白。就是每次新增一个功能以后，进入调试模式试试功能是否能正常运行。这次开发的过程中，一般来说这个方法都能较快的将 Debugger 的代码调试正确。

下一节将会描述调试过程中遇到的一些比较关键的问题。

4.5 遇到的问题

4.5.1 问题一

无法正确暂停。这个就是暂停规则细节一直没有想清楚导致的，经过了 5、6 次的修改，以及漫长编译的等待以后就解决了。

4.5.2 问题二

寄存器无法正常更新。

其实这个问题的出现，才让我意识到了，只流水线暂停而不流水线清空，是不太正确的调试暂停的逻辑。最明显，写寄存器操作会出错。因为你在暂停流水的时候，修改了一个寄存器的值。流水线再度启动的时候，虽然你寄存器的值已经修改，但是位于 EX 已经在更早的时候读取过了寄存器，并保留着未被修改的值进行接下来的运算，这导致的就是你的修改无效。

为了避免这种情况的发生，需要使用流水线暂停的逻辑。这样本来已经到 EX 阶段的指令将会重新执行，就不存在前面的问题了。

当时调试的时候，发现了改寄存器存在这个问题，是通过思考想到而解决的。总的来说，这个问题虽然很严重，但调试的时候相对顺利。

4.6 实现成果

我的 Debugger 模块能稳定地运行，在本次课程也对于我们调试操统起到了极大的帮助。支持的指令主要有：

1. step,next,continue
2. break,hbreak(+ if *)
3. watch,awatch,rwatch
4. bt,finish
5. x,p,disp,set var *=
6. ...

4.7 使用说明

使用方面，并不是一个很麻烦的问题。

首先你需要根据你的 pl2303 串口（片上的那个 USB 串口）绑定的文件，修改 naive-debugger 文件夹中的 naive_mips.c 串口相关的初始化部分。然后 make，在当前目录下得到的 naive-debugger 就是可执行文件。

运行 naive-debugger 后，使用 gdb 的 target remote :1234 连接 naive-debugger（为了方便调试你可以导入符号表），就可以像正常的 GDB 调试代码一样使用了。

5 心得

在本次课程设计中，我有四大收获，一是在做 ucore lab 的阶段，把自己对于操作系统的理解提升了一个档次；第二，是由于本次分工中我主要负责硬件部分，进一步提升了自己的硬件设计、调试能力；然后是，在调试 Flash 的过程中，我学会、并习惯了阅读芯片的 datasheet，这对外设调试是不可或缺的能力；最后，研究片上调试模块、GDB RSP 协议也让我受益匪浅，不但让我独立完成了一个 Debugger 的编码，而且让我对于 GDB 的使用也有了新的理解，这对我以后的动作都有极大的裨益。

当然，本次课程设计中，我们组也存在严重的不足，那就是缺乏沟通。应该建立更好的团队关系，这样我们肯定能在期末的检查中取得更好的成果。