

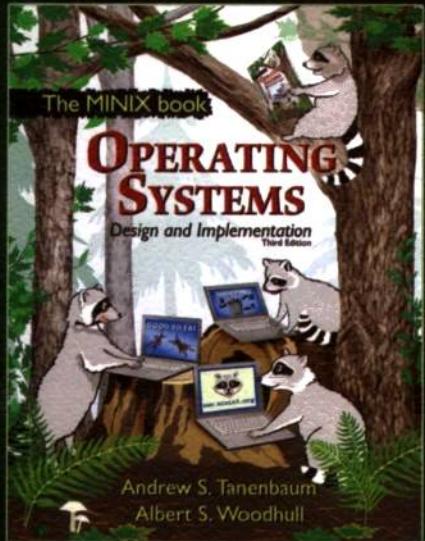
● Tanenbaum's

# 操作系统设计与实现

(第三版) 下册

Operating Systems

Design and Implementation, Third Edition



[美] Andrew S. Tanenbaum 著  
Albert S. Woodhull

陈渝 谌卫军 译  
向勇 审校



电子工业出版社

Publishing House of Electronics Industry

<http://www.phei.com.cn>

经典教材

# 操作系统设计与实现

(第三版) 下册

## Operating Systems

Design and Implementation, Third Edition

### 本书的特点

- **最新且最权威的教材。**本书是世界上最为畅销的操作系统教材。第三版已更新了许多内容，反映了操作系统设计与实现方面的最新进展。
- **理论与实践的完美结合。**首先从总体上介绍操作系统的基本原理和基本概念，然后结合 MINIX 3 系统，深入探讨这些基本原理的具体实现过程，最后再以源代码的形式给出了所有的实现细节。
- **实用性。**MINIX 3 的设计目标是一个实用的、具有高可靠性、灵活性和安全性的系统，能够运行在一些资源有限或者是嵌入式的硬件平台上。系统采用微内核结构，内核代码仅有 4000 行左右，而设备驱动程序等模块则作为普通的用户进程运行，这种结构大大提高了系统的可靠性，对于读者深入掌握操作系统的原理、设计与实现，也是大有裨益的。

### 新特性

- 提供了 MINIX 3 操作系统的光盘。
- 扩充并重组了关于进程与通信方面的内容。
- 修订并增强了关于 CPU 调度、死锁、文件系统可靠性以及安全方面的内容。
- 包含了 150 多道习题。
- 配套资源丰富：[www.minix3.org](http://www.minix3.org) 和 [www.prenhall.com/tanenbaum](http://www.prenhall.com/tanenbaum)。

**Andrew S. Tanenbaum:** 分别在麻省理工学院和加州大学伯克利分校获得学士与博士学位。研究领域包括编译器、操作系统、网络和局域分布式系统、计算机安全等，发表了超过 100 篇论文，并出版了 5 本书籍。Tanenbaum 教授是 ACM 会士、IEEE 会士以及荷兰皇家科学艺术院院士。他还是 1994 年度 ACM Karl V. Karlstrom 杰出教育家奖的获得者，1997 年度 ACM/SIGCSE 计算机科学教育杰出贡献奖的获得者，以及 2002 年度优秀教材奖的获得者。

**Albert S. Woodhull:** 在麻省理工学院获得学士学位，在华盛顿大学获得博士学位。讲授的课程有计算机体系结构、汇编语言程序设计、操作系统和计算机通信。



ISBN 978-7-121-02648-5



9 787121 026485 >



责任编辑：谭海平  
责任美编：毛惠庚



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

定价：49.00 元  
(附光盘 1 张)

国外计算机科学教材系列

# 操作系统设计与实现

( 第三版 )( 下册 )

## Operating Systems

Design and Implementation, Third Edition

[ 美 ] Andrew S. Tanenbaum 著  
Albert S. Woodhull

陈 淦 谌卫军 译

向 勇 审校

电子工业出版社  
Publishing House of Electronics Industry  
北京 · BEIJING

## 内 容 简 介

本书是关于操作系统的权威教材，详细探讨了操作系统的基本原理，包括进程、进程间通信、信号量、管程、消息传递、调度算法、输入/输出、死锁、设备驱动程序、存储管理、调页算法、文件系统设计、安全和保护机制等。大多数关于操作系统的图书均重理论而轻实践，而本书则在这两者之间进行了较好的折中。本册给出了上册中提到的 MINIX 3 操作系统的安装方法以及详细的源代码，并附有光盘。

本书适用于高校计算机专业的学生，也可供程序设计人员、工程技术人员、系统架构师等相关人员参考。

Simplified Chinese edition Copyright © 2006 by PEARSON EDUCATION ASIA LIMITED and Publishing House of Electronics Industry.

Operating Systems: Design and Implementation, Third Edition, ISBN: 0131429388 by Andrew S. Tanenbaum and Albert S. Woodhull. Copyright © 2006.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书中文简体字翻译版由电子工业出版社和 Pearson Education 培生教育出版亚洲有限公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2006-0890

### 图书在版编目（CIP）数据

操作系统设计与实现·下册：第3版 / (美)塔嫩鲍姆 (Tanenbaum, A. S.) 等著；陈渝，谌卫军译。  
北京：电子工业出版社，2007.3

(国外计算机科学教材系列)

书名原文：Operating Systems: Design and Implementation, Third Edition

ISBN 978-7-121-02648-5

I. 操... II. ①塔... ②陈... ③谌... III. 操作系统—教材 IV. TP316

中国版本图书馆 CIP 数据核字 (2006) 第 130357 号

责任编辑：谭海平

印 刷：北京市海淀区四季青印刷厂

装 订：北京牛山世兴印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787 × 1092 1/16 印张：23.25 字数：654 千字

印 次：2007 年 3 月第 1 次印刷

定 价：49.00 元（附光盘 1 张）

凡所购买电子工业出版社的图书有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系电话：(010) 68279077 邮购电话：(010) 88254888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：(010) 88258888

## 出版说明

21世纪初的5至10年是我国国民经济和社会发展的重要时期，也是信息产业快速发展的关键时期。在我国加入WTO后的今天，培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡，是我国面对国际竞争时成败的关键因素。

当前，正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期，为使我国教育体制与国际化接轨，有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材，以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验，翻译出版了“国外计算机科学教材系列”丛书，这套教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时，我们也适当引进了一些优秀英文原版教材，本着翻译版本和英文原版并重的原则，对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上，我们大都选择国外著名出版公司出版的高校教材，如Pearson Education培生教育出版集团、麦格劳-希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者，如道格拉斯·科默(Douglas E. Comer)、威廉·斯托林斯(William Stallings)、哈维·戴特尔(Harvey M. Deitel)、尤利斯·布莱克(Uyless Black)等。

为确保教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士，也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括对所选教材进行全面论证；选择编辑时力求达到专业对口；对排版、印制质量进行严格把关。对于英文教材中出现的错误，我们通过与作者联络和网上下载勘误表等方式，逐一进行了修订。

此外，我们还将与国外著名出版公司合作，提供一些教材的教学支持资料，希望能为授课老师提供帮助。今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

## 教材出版委员会

主任	杨芙清	北京大学教授 中国科学院院士 北京大学信息与工程学部主任 北京大学软件工程研究所所长
委员	王 珊	中国人民大学信息学院院长、教授
	胡道元	清华大学计算机科学与技术系教授 国际信息处理联合会通信系统中国代表
	钟玉琢	清华大学计算机科学与技术系教授、博士生导师 清华大学深圳研究生院信息学部主任
	谢希仁	中国人民解放军理工大学教授 全军网络技术研究中心主任、博士生导师
	尤晋元	上海交通大学计算机科学与工程系教授 上海分布计算技术中心主任
	施伯乐	上海国际数据库研究中心主任、复旦大学教授 中国计算机学会常务理事、上海市计算机学会理事长
	邹 鹏	国防科学技术大学计算机学院教授、博士生导师 教育部计算机基础课程教学指导委员会副主任委员
	张昆藏	青岛大学信息工程学院教授

# 译者序

Andrew S. Tanenbaum是著名的计算机科学家、教育家，荷兰皇家科学艺术院院士，也是IEEE会士和ACM会士，目前供职于荷兰阿姆斯特丹Vrije大学。他在操作系统、分布式系统和计算机网络等领域都有很深的造诣，曾多次获奖，包括1994年度ACM Karl V. Karlstrom杰出教育家奖、1997年度ACM CSE计算机科学教育杰出贡献奖、2002年度TAA优秀教材奖和2003年度TAA McGuffey奖。

20世纪80年代，出于教学工作的需要，Tanenbaum教授开发了一个小巧、完整、开放源代码、UNIX兼容的操作系统MINIX，使学生可以通过剖析这个“麻雀虽小，五脏俱全”的系统，来研究其内部的运作机理。为了便于学习，他还出版了相应的教材，即本书的第一版。经过20多年的发展，MINIX系统在许多方面得到了改进，如对现代主流硬件设备的支持、对POSIX标准的支持、微内核系统结构等；与之相对应，本书也不断推陈出新，在2006年出版了第三版。

本书的最大特点就是理论与实践的完美结合。在多年的教学实践中，我们深刻地体会到，对于操作系统这样一门实用性和实践性很强的课程，如果只是单纯地介绍它的基本原理和基本概念，很难有非常理想的教学效果。一个连进程的创建函数都没有用过的人，很难想象他能对进程与线程之间的区别有真正的了解。同样，一个没有分析过内存分配源代码的人，也很难对虚拟存储管理有太多深入的理解。而本书的出现则弥补了这个缺陷，在理论与实践之间，搭起了一座桥梁。本书涵盖了操作系统课程的所有内容，包括进程管理、存储管理、文件系统和设备管理等。对于每一个章节，在组织结构上采用了从浅到深、从抽象到具体、从宏观到细节的讲授方式。首先从总体上介绍操作系统的根本原理和基本概念，然后结合MINIX 3系统，深入探讨这些基本原理的具体实现过程，最后再以源代码的形式给出了所有的实现细节。通过这种自顶向下、逐步求精的学习过程，使读者能够做到融会贯通。在面对抽象、枯燥的理论时，能够用技术实现来加以印证、加深理解；在面对复杂、繁琐的源代码时，能够用理论思想来进行指导。相信这样的一种学习模式，对于读者深入掌握操作系统的原理、设计与实现，是大有裨益的。

本书的另一个特点是实用性。如果说MINIX 1和MINIX 2还主要是用于教学目的，那么MINIX 3则完全不同。它的设计目标是一个实用的、具有高可靠性、灵活性和安全性的系统，能够运行在一些资源有限或者是嵌入式的硬件平台上。系统采用微内核结构，内核代码仅有4000行左右，而设备驱动程序等模块则作为普通的用户进程运行，这种结构大大提高了系统的可靠性，读者只要加以修改，就可以移植到自己的硬件平台上。

基于上述原因，我们认为翻译本书、把它介绍给国内的读者是一件非常有意义的事情，衷心希望我们付出的劳动能对国内的操作系统教学和实践有所帮助和促进。

本书的第2章、第3章由陈渝翻译并统稿，第1章、第4章、第5章、第6章由谌卫军翻译，向勇对全书进行了审校。在整个翻译过程中，清华大学计算机系和软件学院的师生给予了帮助，并且在计算机系和软件学院的本科生的操作系统课程中进行了试用，许多学生提出了很好的建议，在此向他们表示衷心的感谢。

译者

2006年11月于清华园

# 作者简介

## Andrew S. Tanenbaum

Andrew S. Tanenbaum 分别在麻省理工学院和加州大学伯克利分校获得学士与博士学位。现任荷兰阿姆斯特丹 Vrije 大学计算机科学教授并领导着一个计算机系统研究小组。到 2005 年 1 月卸任为止，他担任计算与成像高级学院院长一职已有 12 年。

Tanenbaum 过去的研究领域包括编译器、操作系统、网络和局域分布式系统，而现在的研究方向则主要为计算机安全，尤其是操作系统、网络以及分布式系统的安全。在所有这些研究领域，Tanenbaum 发表了超过 100 篇论文，并出版了 5 本书籍。

Tanenbaum 教授还编写了大量软件。他是 Amsterdam Compiler Kit（一种广泛使用的、用于编写可移植编译器以及 MINIX 的工具集）的主要开发者，而该系统则是 Linux 诞生的灵感与基础。与他的博士生及程序员一起，他帮助设计了 Amoeba 分布式操作系统（一个基于微内核的、高性能局域分布式操作系统）。此后，他是 Globe（一个可处理 10 亿用户的广域分布式操作系统）的设计者之一。所有这些软件现在均可在互联网上免费获得。

他的博士生在毕业后均取得了很大的成绩，他为此感到非常骄傲。

Tanenbaum 教授是 ACM 会士、IEEE 会士以及荷兰皇家科学艺术院院士。他还是 1994 年度 ACM Karl V. Karlstrom 杰出教育家奖的获得者，1997 年度 ACM/SIGCSE 计算机科学教育杰出贡献奖的获得者，以及 2002 年度优秀教材奖的获得者。2004 年，他被推选为荷兰皇家学会的五位新学会教授之一。他的主页地址为 <http://www.cs.vu.nl/~ast/>。

## Albert S. Woodhull

Albert S. Woodhull 在麻省理工学院获得学士学位，在华盛顿大学获得博士学位。在入读麻省理工学院时，他希望能成为一位电气工程师，但最后却成了一位生物学家。20 多年来，他一直工作于麻省 Amherst 的 Hampshire 自然科学院。当微型计算机慢慢多起来的时候，作为使用电子检测仪器的生物学家，他开始使用微型计算机。他给学生开设的仪器检测方面的课程逐渐演变为计算机接口和实时程序设计课程。

Woodhull 博士对教学和科学技术的发展有浓厚的兴趣，在进入研究生院之前，他曾在尼日利亚教过两年中学，近年来他曾几次利用自己的假期在尼加拉瓜讲授计算机科学。

他对计算机作为电子系统，以及计算机与其他电子系统的相互配合很感兴趣。他最喜欢讲授的课程有计算机体系结构、汇编语言程序设计、操作系统和计算机通信。他还为开发电子器件及相关软件担任顾问。

在学术之外，Woodhull 还有不少兴趣，包括各种户外活动、业余无线电制作和读书。他还喜欢旅游和学习别国语言。他的主页放在一台运行 MINIX 的机器上，地址为 <http://minix1.hampshire.edu/asw/>。

# 序　　言

大多数关于操作系统的图书均重理论而轻实践，而本书则在这两者之间进行了较好的折中。本书详细探讨了操作系统的基本原理，包括进程、进程间通信、信号量、管程、消息传递、调度算法、输入/输出、死锁、设备驱动程序、存储管理、调页算法、文件系统设计、安全和保护机制等；此外，还详细讨论了一个特殊的操作系统 MINIX 3（一个与 UNIX 兼容的操作系统），甚至提供了该系统的源代码，以便于读者仔细研究。这种安排不仅可让读者了解操作系统的基本原理，而且可让读者了解到这些基本原理是如何应用到真实的操作系统中去的。

自本书第一版于 1987 年推出以来，操作系统课程的教学方式已有了较小的革新。在此之前，大多数课程都只讲授理论知识。随着 MINIX 的出现，许多学校开始开设实验课程，以便学生仔细研究操作系统，了解其内部工作机理。对于这种教学方式，我们相当欣慰并希望能继续加强这种趋势。

MINIX 在推出以来的 10 年里经历了许多变化。最初的代码是为基于 8088 的有着 256 KB 内存和两个磁盘驱动器的 IBM PC 机（没有硬盘）开发的，它基于 UNIX V7。随着时间的推移，MINIX 在许多方面已得到了改进，例如它支持有着大内存和多个硬盘的 32 位保护模式的机器，而且它不再基于 UNIX V7，而基于国际 POSIX 标准（IEEE 1003.1 和 ISO 9945-1）。与此同时，UNIX 中添加了许多新特性，在我们看来，这些新添加的特性可能很多，但在某些人看来这些新特性还不够，因而导致了 Linux 的出现。此外，MINIX 还被移植到了许多其他的平台，包括 Macintosh, Amiga, Atari 和 SPARC。本书的第二版于 1997 年出版并在大学中广泛使用。

MINIX 仍在得到人们的欢迎，这可以通过 Google 上对 MINIX 的搜索次数看出。

本书的第三版已有了很大的变化。我们修订了关于原理的几乎所有内容，并添加了许多新内容，但主要变化是对新操作系统 MINIX 3 的讨论以及本书中包含的新代码。尽管仍松散地基于 MINIX 2，但 MINIX 3 在几个主要的方面几乎完全不同。

设计 MINIX 3 的目的基于如下事实：操作系统正变得越来越大、越来越慢以及越来越不可靠。与其他电子设备如电视、手机以及 DVD 播放器相比，操作系统变得越来越容易崩溃，加之具有许多特性和选项，实际上几乎没有人能完全理解它们，或将它们管理得很好。当然，计算机病毒、蠕虫、间谍软件以及其他形式的恶意程序也已变得越来越猖獗，这对操作系统无疑会造成较大的影响。

在很大程度上，这些问题均是由当前操作系统中的基本设计缺陷引起的：模块性的缺失。整个操作系统一般会由几百万行 C/C++ 代码组成，这些代码被编译到了一个在内核态下运行的巨大可执行程序中。在几百万行代码中，即使只有一行代码存在缺陷，也会导致系统发生故障。使所有这些代码均正确是不可能的，尤其是当 70% 的代码是由设备驱动程序组成时，因为这些设备驱动程序是由第三方编写的，而这已超出了操作系统编写者的控制范围。

通过使用 MINIX 3，我们证明了在设计操作系统时，并不是只有整体式设计这一种方法。MINIX 3 内核仅包含有约 4000 行可执行代码，而不是 Windows, Linux, Mac OS X 或 FreeBSD 的数百万行代码。系统中的其他内容，包括所有的设备驱动程序（除时钟驱动程序外），是一个模块化的用户模式进程的小集合，对于每个进程，我们只关注其作用以及与之通信的其他进程。

尽管 MINIX 3 仍在改进，但我们相信这个将操作系统构建为高度封装的用户模式进程的模型完全可以在将来用于构建更为可靠的系统。MINIX 3 适用于小型 PC 机（如那些可在第三世界国家

找到的机器以及嵌入式系统，这些机器的资源均有限）。无论如何，这种设计可使得学生更易于了解操作系统的工作原理，而不必去试图研究巨大的整体式操作系统。

本书附带的 CD-ROM 是一张可引导 CD。读者可将该 CD-ROM 放到驱动器中，重新启动计算机，之后 MINIX 3 会在几秒钟后显示登录提示符。读者可登录为 root 用户来启动系统，而不用将 MINIX 3 首先安装到硬盘中。当然，MINIX 3 也可安装到硬盘上。详细的安装信息请参阅附录 A。

像上面建议的那样，MINIX 3 正在快速发展，新版本正在频繁地推出。要下载当前的 CD-ROM 映像文件来刻录光盘，请进入 MINIX 的官方网站 [www.minix3.org](http://www.minix3.org)。该网站还包含有大量的新软件、文献，以及关于 MINIX 3 开发的新闻。关于 MINIX 3 的讨论以及频繁问及的问题，读者可进入 USENET 新闻组 *comp.os.minix* 了解详情。没有新闻阅读器的人们可按网站 <http://groups.google.com/group/comp.os.minix> 上的说明操作。

MINIX 3 除了可安装到硬盘上来运行外，也可在几个 PC 模拟器上运行。网站的主页上列出了些模拟器。

使用本书作为教材的教师，可通过 Prentice Hall 出版公司在当地的机构来获得习题解答（索取方式请参阅书后的“教学支持说明”）。此外，本书有其自己的配套网站，网址为 [www.prenhall.com/tanenbaum](http://www.prenhall.com/tanenbaum)。

在本书的编写过程中，我们有幸得到了许多人的帮助。Ben Gras 和 Jorrit Herder 在时间仓促的情况下，完成了新版本操作系统的大部分编程工作；此外，他们还阅读了本书的初稿并做了许多有用的注释。在此，我们要对他们深表谢意。

Kees Bot 曾为 MINIX 2 做了许多工作，这为我们开发 MINIX 3 奠定了很好的基础。Kees 为 MINIX 的最初版本至版本 2.0.4 编写了大量的代码，修复了缺陷，并回答了许多问题。Philip Homburg 也编写了大量的代码，并在其他方面也给予了我们大量的帮助，尤其是在初稿的审读方面。

当然，MINIX 的推出还得到了其他许多人的帮助，在此我们一并表示致谢。

一些人阅读了部分初稿并提出了修订建议。在此要特别感谢 Gojko Babic, Michael Crowley, Joseph M. Kizza, Sam Kohn Alexander Manov 和 Du Zhang。

最后，我们要感谢我们的家庭。Suzanne 已是第 16 次在我埋头写作时给予我支持，Barbara 已是第 15 次，而 Marvin 已是 14 次。他们的支持和爱心对我非常重要 (AST)。

Al's Barbara 已是第二次在我写作时给我支持。若没有她的支持、耐心，我们是不可能完成这项工作的。我的儿子 Gordon 一直是一位有耐心的倾听者，能得到儿子的理解与支持，对我而言无疑十分欣慰。最后，教父 Zain 的生日恰好就是 MINIX 3 的发布日期，也许某一天他会对此感到高兴的 (ASW)。

Andrew S. Tanenbaum

Albert S. Woodhull

# 关于 MINIX 3 CD

## 系统需求

下面是安装光盘中提供的软件时的最小系统需求。

## 硬件

MINIX 3 操作系统要求如下的硬件：

- Pentium 或兼容处理器的 PC
- 16 MB 或更大的 RAM
- 200 MB 的可用磁盘空间
- IDE CD-ROM 驱动器
- IDE 硬盘

不支持串行 ATA, USB 和 SCSI 硬盘。对于可选择的其他配置, 请访问 <http://www.minix3.org>。

## 软件

MINIX 3 OS 是一个操作系统。如果读者希望保留原有的操作系统和数据（建议）并创建一个双重引导机器，那么需要对硬盘分区。这时可使用如下软件之一来进行分区：

- Partition Magic (<http://www.powerquest.com/partitionmagic>)
- Partition Resizer (<http://www.zeleps.com>)
- 按网页 <http://www.minix3.org/partitions.html> 上的步骤操作

## 安装

在未连接到 Internet 的情况下可以完成安装, 但一些高级文档仅联机出现在网站 <http://www.minix3.org> 上。完成安装的说明在 CD 上以 Adobe Acrobat PDF 格式提供。

## 产品支持

关于该光盘中 MINIX 软件的进一步技术信息, 请访问 MINIX 的官方网站 <http://www.minix3.org>。

# 目 录

附录 A 安装 MINIX 3 .....	1
附录 B MINIX 源码 .....	7
附录 C 文件索引 .....	356

# 附录 A 安装 MINIX 3

- A.1 准备**
- A.2 启动**
- A.3 安装到硬盘**
- A.4 测试**
- A.5 使用模拟器**

本附录解释如何安装 MINIX 3。完全安装 MINIX 3 的要求如下：奔腾（或兼容的）处理器，至少 16 MB 的 RAM，1 GB 的硬盘可用空间，一台 IDE CD-ROM 以及一个 IDE 硬盘。最小安装（无命令源代码）需要 8 MB 的 RAM 和 50 MB 的硬盘空间。当前不支持串行 ATA、USB 和 SCSI 磁盘。对于 USB CD-ROM，请参阅网站 [www.minix3.org](http://www.minix3.org)。

## A.1 准备

如果已有 CD-ROM（如取自本书），那么可跳过步骤 1 和步骤 2，但建议读者最好查阅网站 [www.minix3.org](http://www.minix3.org)，以了解是否有更新的版本可用。如果想在模拟器而不是本地机器上运行 MINIX 3，则要首先参阅第五部分。如果没有 IDE CD-ROM，则要么获得一个专用的 USB CD-ROM 启动映像，要么使用一个模拟器。

### 1. 下载 MINIX 3 CD-ROM 映像

读者可从 MINIX 3 的网站 [www.minix3.org](http://www.minix3.org) 上下载 MINIX 3 CD-ROM 映像。

### 2. 下载可启动的 MINIX 3 CD-ROM

解压缩下载后的文件。这时会得到带有扩展名.iso 的一个 CD-ROM 映像文件以及本手册。扩展名为.iso 的文件是一个逐位 CD-ROM 映像，将其刻录在 CD-ROM 上是为了使该 CD-ROM 成为可启动的 CD-ROM。

如果正在使用 *Easy CD Creator 5* 刻录程序，那么可以从 File 菜单中选择 Record CD from CD image，并在出现的对话框中将文件类型从.cif 更改为.iso。选择该映像文件并单击 Open，然后单击 Start Recording。

如果正使用 *Nero Express 5* 刻录程序，那么可选择 Disc Image or Saved Project 并将类型更改为 Image Files，选择该映像文件并单击 Open。选择自己的 CD 刻录机并单击 Next。

如果正在运行 Windows XP 且没有 CD-ROM 刻录程序，那么可进入网页 [alexfeinman.brinkster.net/isorecorder.htm](http://alexfeinman.brinkster.net/isorecorder.htm) 下载一个免费的程序并使用它来创建一个 CD 映像。

### 3. 确定所拥有的以太网芯片

MINIX 3 支持几种以太网芯片，包括 Intel Pro/100, RealTek 8029 和 8139, AMD LANCE 以及几种 3COM 芯片。在安装过程中，系统将会询问读者的以太网芯片是哪一种。请读者查询文档以确定这一点。另外，如果读者正在使用 Windows，那么可按如下方式进入设备管理器：

Windows 2000: Start > Settings > Control Panel > System > Hardware > Device Manager

Windows XP: Start > Control Panel > System > Hardware > Device Manager

其中 System 要求双击；其他图标或菜单项要求单击。单击 + 以展开到出现 Network adapters，这时会看到所拥有的以太网芯片类型。如果没有支持的芯片，那么仍可运行 MINIX 3，只是这时系统不支持以太网。

#### 4. 对硬盘进行分区

如果愿意，可从CD-ROM启动计算机，MINIX 3将会启动，但如果要做一些有意义的事情，则必须在硬盘上为其创建一个分区。但在进行分区之前，为安全起见，一定要将自己的数据备份到外部媒介如CD-ROM或DVD中。读者的文件是有价值的，一定要保护好它们。

如果不知道如何对磁盘分区或是经验不足，那么这里建议读者在[www.minix3.org/doc/partitions.html](http://www.minix3.org/doc/partitions.html)处阅读关于磁盘分区的联机教程。如果已经知道如何管理分区，那么创建一个至少50 MB的连续可用磁盘空间，在需要命令源代码的情况下，则至少要创建一个1 GB大小的磁盘空间。如果不知道如何管理分区但又有一个如*Partition Magic*这样的分区程序，那么可以使用它来创建一个有着可用磁盘空间的分区。注意，一定要确保至少有一个主分区（如Master Boot Record插槽）是可用的。MINIX 3安装脚本将指导读者在可用空间中创建一个MINIX分区，这个分区可在第一个或第二个IDE磁盘上。

如果正在运行Windows 95, 98, Me或2000，且磁盘由单个FAT分区组成，则可使用CD-ROM上的*presz134.exe*程序（也可在网站zeleps.com处获得）来调整其大小，以便为MINIX腾出空间。在其他情形下，请阅读上面给出的联机教程。

如果读者的磁盘大于128 GB，则MINIX 3分区必须完全位于前128 GB中（因为磁盘块是按这种方法寻址的）。

**警告：**如果在磁盘分区过程中出现了错误，那么可能会丢失磁盘上的所有数据，因此一定要在分区前将磁盘上的数据备份到CD-ROM或DVD中。在进行磁盘分区时一定要小心。

## A.2 启动

现在，读者应已在磁盘上分配了一些可用空间。如果现在还未做到这一点，那么请立即这样做，除非读者愿意将一个已存在的分区转换为MINIX 3分区。

### 1. 从CD-ROM启动

将CD-ROM插入光盘驱动器中并从它启动计算机。如果有16 MB或以上的内存，可以选择Regular；如果仅有8 MB的内存，则选择Small。如果计算机是从硬盘而不是从CD-ROM启动，那么请再次启动并进入BIOS设置程序来更改启动设备的顺序，即将CD-ROM放在硬盘的前面。

### 2. 作为root用户登录

当登录提示符出现时，请作为root用户登录。在成功地作为root用户登录后，会看到命令提示符（#）。这时表示读者正在完整地运行MINIX 3。如果键入

```
ls/usr/bin | more
```

则会看到可用的软件。按空格键并滚动列表。为了解程序foo的功能，可键入

```
man foo
```

也可自[www.minix3.org/manpages](http://www.minix3.org/manpages)处查阅相关的手册页面。

### 3. 启动安装脚本

要在硬盘上开始MINIX 3的安装，可键入

```
setup
```

在该命令及所有其他命令之后，一定要按回车键。当安装脚本在屏幕上以一个分号结束时，按回车键继续。如果屏幕突然变得空白，请按 CTRL-F3 组合键来滚动选择软件（仅在非常旧的计算机上需要这样做）。注意，CTRL-F3 意味着按下 CTRL 键的同时按下 F3 键。

## A.3 安装到硬盘

以下这些步骤相应于读者屏幕上的步骤。

### 1. 选择键盘类型

在系统要求读者选择自己国家的键盘时，请这样做。该步骤和其他步骤在方括号中提供一个默认的选择。如果用户同意默认的选择，可按回车键。在多数步骤中，默认的选择对初学者来说都是较好的选择。

### 2. 选择以太网芯片

现在系统将询问用户想要安装哪种可用的以太网驱动程序。请选择一种。

### 3. 基本的最小或完全安装

如果磁盘空间很紧张，那么可选择 M 来进行最小安装，最小安装会安装所有的二进制码以及系统的源码，但不会安装命令的源码。对于基本的系统，50 MB 可用空间已足够了。如果有 1 GB 或大于 1 GB 的可用空间，那么可以选择 F 来进行完全安装。

### 4. 为 MINIX 3 创建或选择一个分区

系统首先会询问读者是否精通于为 MINIX 3 进行磁盘分区。如果精通，则可进入分区程序并为用户提供编辑主引导记录的全部权限。如果不精通，则可按回车键接受默认的操作，默认操作会为 MINIX 3 自动地逐步格式化磁盘分区。

#### 4.1 选择一个磁盘来安装 MINIX 3

一个IDE控制器最多可接四个磁盘。安装脚本现在会搜索每个磁盘。这时可忽略任何错误消息。在列出驱动器后，选择其中的一个驱动器并确认自己的选择。如果有两个硬盘且决定将 MINIX 3 安装到第二个硬盘上并在第二硬盘启动时出现了问题，可参阅 [www.minix3.org/doc/using2disks.html](http://www.minix3.org/doc/using2disks.html)。

#### 4.2 选择一个磁盘区域

现在选择一个区域来安装 MINIX 3。这时有三种选择：

- (1) 选择一个可用的区域。
- (2) 选择一个将要覆盖的区域。
- (3) 删除一个分区以腾出空间并与相邻的可用空间合并。

对于选择(1)和选择(2)，键入磁盘区域编号。对于选择(3)，键入

`delete`

然后在系统询问时给出区域编号。这个区域将被覆盖且其先前的内容将永远消失。

#### 4.3 确认选择

现在将进入不可逆转的阶段。这时系统会询问是否继续。如果继续，则选择的区域中的数据将会永远丢失。如果确定，可键入

`yes`

然后按回车键。要在不更改分区表的情况下退出安装脚本，可按 CTRL-C 组合键。

### 5. 重新安装选项

如果选择一个已有的 MINIX 3 分区，这时会出现两个选项，即 Full install 和 Reinstall。选择前一个选项时，会删除分区中的所有内容；选择后一个选项时，则不影响已存在的 */home* 分区。这种设计意味着用户可将自己的个人文件放到 */home* 中并重新安装 MINIX 3 的一个新版本，而不会丢失读者的个人文件。

### 6. 选择 */home* 的大小

选取的分区将分为三个子分区，即 *root*, */usr* 和 */home*。后者用于存放用户自己的个人文件。请指定为自己的文件保留多少分区的空间。系统会要求用户对选择进行确认。

### 7. 选择块大小

系统支持 1 KB, 2 KB, 4 KB 和 8 KB 的块大小。如果要使用一个大于 4 KB 的块大小，则必须修改一个常量并重新编译系统。如果用户的内存是 16 MB 或更大，可使用默认值（4 KB）；否则，请使用 1KB。

### 8. 等待坏块检测

安装脚本现在将扫描每个分区中的坏磁盘块。这需要几分钟时间，对于较大的分区可能要 10 分钟或更长的时间，请耐心等待。如果读者确认无坏块，则可按 CTRL-C 组合键中止对每个分区的扫描。

### 9. 等待复制文件

在扫描完成后，文件将会自动地从 CD-ROM 中复制到硬盘中。每个文件都将声明被复制。在复制完成后，MINIX 3 的安装也就完成了。键入

```
shutdown
```

关闭系统。要总是按照这种方法来停止 MINIX 3，以避免数据丢失，因为 MINIX 3 会在 RAM 磁盘上保留一些文件，尽管这些文件已复制到了硬盘中。

## A.4 测试

本节描述如何测试安装、在修改安装后如何重建系统以及如何在以后启动它。为开始测试，首先要启动新的 MINIX 3 系统。例如，如果使用了控制器 0、磁盘 0 和分区 3，则可键入

```
boot c0d0p3
```

并作为 root 用户登录。在很少见的条件下，BIOS 可见的驱动器号（由引导监控程序使用）可能与 MINIX 3 使用的驱动器号不一致。首先请试用安装脚本声明的那个驱动器号。这时可创建一个 root 密码。要获得帮助，请参阅 *man passwd*。

### 1. 编译测试套件

要测试 MINIX 3，可在命令提示符 (#) 下键入

```
cd /usr/src/test  
make
```

并等待，直到它完成了所有的 40 个编译。按 CTRL-D 组合键退出登录。

## 2. 运行测试套件

为测试系统，登录为 bin（要求）并键入

```
cd /usr/src/test  
.run
```

来运行测试程序。它们应能正确地运行，但可能需要 20 分钟至 1 小时的时间，具体要视机器的快慢而定。注意，在作为 root 运行时，需要编译测试套件，但要作为 bin 执行，以了解 stupid 位是否正确地工作。

## 3. 重建整个操作系统

如果所有的测试工作正确，那么现在就可以重建系统。这样做是不必要的，因为它已预建完毕；但如果读者计划修改系统，则需要知道如何重建系统。此外，重建系统可较好地测试系统是否工作正常。键入

```
cd /usr/src/tools  
make
```

查看各种可用的选项。现在，通过键入

```
su  
make clean  
time make image
```

来建立一个新的可引导映像。这只是重建操作系统，包括所有的内核态和用户态部分。所需要的时间不会很长。如果有一个软盘驱动器，则可将一张格式化后的软盘插入到其中并键入

```
make fdboot
```

来制作一张可引导软盘，以备将来使用。

当系统询问完整的路径时，可键入

```
fd0
```

这种方法当前不适用于 USB 软盘，因为现在还没有 MINIX 3 USB 软盘驱动程序。要更新当前已安装到硬盘上的引导映像，可键入

```
make hdboot
```

## 4. 关闭与重启新系统

要启动新系统，首先要键入

```
shutdown
```

来关闭系统。该命令会保存一些文件并回到 MINIX 3 引导监控程序中。要大致了解引导监控程序的功能，可键入

```
help
```

对于更加详细的信息，请参阅 [www.minix3.org/manpages/man8/boot.8.html](http://www.minix3.org/manpages/man8/boot.8.html)。现在可取出 CD-ROM 或软盘并关闭计算机。

## 5. 后续启动

如果有一个软盘驱动器，则启动 MINIX 3 的最简方法是插入新的引导软盘并打开计算机的电源。这只需要几秒钟时间。也可从 MINIX 3 CD-ROM 引导，登录为 bin 并键入

```
shutdown
```

以返回到 MINIX 3 引导监控程序。现在键入

```
boot c0d0p0
```

以从控制器 0、驱动器 0、分区 0 上的操作系统映像文件启动。当然，如果将 MINIX 3 放到驱动器 0 分区 1 上，那么可使用

```
boot c0d0p1
```

依次类推。

第三种启动方法是使 MINIX 3 分区成为活动分区，并使用 MINIX 3 引导监控程序来启动 MINIX 3 或任何其他操作系统。详细内容请参阅 [www.minix3.org/manpages/man8/boot.8.html](http://www.minix3.org/manpages/man8/boot.8.html)。

最后，第四种方法是安装一个多启动装载程序，如 LILO 或 GRUB ([www.gnu.org/software/grub](http://www.gnu.org/software/grub))。然后，就可以很容易地启动任何操作系统。多启动装载程序的讨论超出了本指南的范围，读者可在 [www.minix3.org/doc](http://www.minix3.org/doc) 中找到关于该主题的一些信息。

## A.5 使用模拟器

运行 MINIX 3 的一种完全不同的方法是在另一个操作系统之上运行它。各种虚拟机、模拟器和仿真器可用于此目的。下面是一些最为流行的模拟器：

- VMware ([www.vmware.com](http://www.vmware.com))
- Bochs ([www.bochs.org](http://www.bochs.org))
- QEMU ([www.qemu.org](http://www.qemu.org))

对于上面的每个模拟器，请读者参阅相关的文档。在模拟器上运行一个程序类似于在一台实际的机器上运行它，因此读者应返回到第一部分，获取最新的 CD-ROM 并在此处继续。

## 附录 B MINIX 源码

```
+++++
      include/ansi.h
+++++  

00000 /* The <ansi.h> header attempts to decide whether the compiler has enough
00001   * conformance to Standard C for Minix to take advantage of. If so, the
00002   * symbol _ANSI is defined (as 31459). Otherwise _ANSI is not defined
00003   * here, but it may be defined by applications that want to bend the rules.
00004   * The magic number in the definition is to inhibit unnecessary bending
00005   * of the rules. (For consistency with the new '#ifdef _ANSI' tests in
00006   * the headers, _ANSI should really be defined as nothing, but that would
00007   * break many library routines that use "#if _ANSI".)
00008
00009   * If _ANSI ends up being defined, a macro
00010   *
00011   *     _PROTOTYPE(function, params)
00012
00013   * is defined. This macro expands in different ways, generating either
00014   * ANSI Standard C prototypes or old-style K&R (Kernighan & Ritchie)
00015   * prototypes, as needed. Finally, some programs use _CONST, _VOIDSTAR etc
00016   * in such a way that they are portable over both ANSI and K&R compilers.
00017   * The appropriate macros are defined here.
00018 */
00019
00020 #ifndef _ANSI_H
00021 #define _ANSI_H
00022
00023 #if __STDC__ == 1
00024 #define _ANSI           31459 /* compiler claims full ANSI conformance */
00025 #endif
00026
00027 #ifdef __GNUC__
00028 #define _ANSI           31459 /* gcc conforms enough even in non-ANSI mode */
00029 #endif
00030
00031 #ifdef _ANSI
00032
00033 /* Keep everything for ANSI prototypes. */
00034 #define _PROTOTYPE(function, params)    function params
00035 #define _ARGS(params)                  params
00036
00037 #define _VOIDSTAR      void *
00038 #define _VOID          void
00039 #define _CONST         const
00040 #define _VOLATILE     volatile
00041 #define _SIZET         size_t
00042
00043 #else
00044
00045 /* Throw away the parameters for K&R prototypes. */
00046 #define _PROTOTYPE(function, params)    function()
00047 #define _ARGS(params)                  ()
00048
00049 #define _VOIDSTAR      void *
00050 #define _VOID          void
00051 #define _CONST         const
00052 #define _VOLATILE     volatile
00053 #define _SIZET         int
00054
```

```

00055 #endif /* _ANSI */
00056
00057 /* This should be defined as restrict when a C99 compiler is used. */
00058 #define _RESTRICT
00059
00060 /* Setting any of _MINIX, _POSIX_C_SOURCE or _POSIX2_SOURCE implies
00061 * _POSIX_SOURCE. (Seems wrong to put this here in ANSI space.)
00062 */
00063 #if defined(_MINIX) || _POSIX_C_SOURCE > 0 || defined(_POSIX2_SOURCE)
00064 #undef _POSIX_SOURCE
00065 #define _POSIX_SOURCE 1
00066 #endif
00067
00068 #endif /* ANSI_H */

+++++
include/limits.h
++++

00100 /* The <limits.h> header defines some basic sizes, both of the language types
00101 * (e.g., the number of bits in an integer), and of the operating system (e.g.
00102 * the number of characters in a file name).
00103 */
00104
00105 #ifndef _LIMITS_H
00106 #define _LIMITS_H
00107
00108 /* Definitions about chars (8 bits in MINIX, and signed). */
00109 #define CHAR_BIT 8 /* # bits in a char */
00110 #define CHAR_MIN -128 /* minimum value of a char */
00111 #define CHAR_MAX 127 /* maximum value of a char */
00112 #define SCHAR_MIN -128 /* minimum value of a signed char */
00113 #define SCHAR_MAX 127 /* maximum value of a signed char */
00114 #define UCHAR_MAX 255 /* maximum value of an unsigned char */
00115 #define MB_LEN_MAX 1 /* maximum length of a multibyte char */
00116
00117 /* Definitions about shorts (16 bits in MINIX). */
00118 #define SHRT_MIN (-32767-1) /* minimum value of a short */
00119 #define SHRT_MAX 32767 /* maximum value of a short */
00120 #define USHRT_MAX 0xFFFF /* maximum value of unsigned short */
00121
00122 /* _EM_WSIZE is a compiler-generated symbol giving the word size in bytes. */
00123 #define INT_MIN (-2147483647-1) /* minimum value of a 32-bit int */
00124 #define INT_MAX 2147483647 /* maximum value of a 32-bit int */
00125 #define UINT_MAX 0xFFFFFFFF /* maximum value of an unsigned 32-bit int */
00126
00127 /*Definitions about longs (32 bits in MINIX). */
00128 #define LONG_MIN (-2147483647L-1)/* minimum value of a long */
00129 #define LONG_MAX 2147483647L /* maximum value of a long */
00130 #define ULONG_MAX 0xFFFFFFFFL /* maximum value of an unsigned long */
00131
00132 #include <sys/dir.h>
00133
00134 /* Minimum sizes required by the POSIX P1003.1 standard (Table 2-3). */
00135 #ifdef _POSIX_SOURCE /* these are only visible for POSIX */
00136 #define _POSIX_ARG_MAX 4096 /* exec() may have 4K worth of args */
00137 #define _POSIX_CHILD_MAX 6 /* a process may have 6 children */
00138 #define _POSIX_LINK_MAX 8 /* a file may have 8 links */
00139 #define _POSIX_MAX_CANON 255 /* size of the canonical input queue */
00140 #define _POSIX_MAX_INPUT 255 /* you can type 255 chars ahead */
00141 #define _POSIX_NAME_MAX DIRSIZ /* a file name may have 14 chars */
00142 #define _POSIX_NGROUPS_MAX 0 /* supplementary group IDs are optional */
00143 #define _POSIX_OPEN_MAX 16 /* a process may have 16 files open */
00144 #define _POSIX_PATH_MAX 255 /* a pathname may contain 255 chars */
00145 #define _POSIX_PIPE_BUF 512 /* pipes writes of 512 bytes must be atomic */
00146 #define _POSIX_STREAM_MAX 8 /* at least 8 FILEs can be open at once */
00147 #define _POSIX_TZNAME_MAX 3 /* time zone names can be at least 3 chars */

```

```

00148 #define _POSIX_SSIZE_MAX 32767 /* read() must support 32767 byte reads */
00149
00150 /* Values actually implemented by MINIX (Tables 2-4, 2-5, 2-6, and 2-7). */
00151 /* Some of these old names had better be defined when not POSIX. */
00152 #define _NO_LIMIT 100 /* arbitrary number; limit not enforced */
00153
00154 #define NGROUPS_MAX 0 /* supplemental group IDs not available */
00155 #define ARG_MAX 16384 /* # bytes of args + environ for exec() */
00156 #define CHILD_MAX _NO_LIMIT /* MINIX does not limit children */
00157 #define OPEN_MAX 20 /* # open files a process may have */
00158 #define LINK_MAX SHRT_MAX /* # links a file may have */
00159 #define MAX_CANON 255 /* size of the canonical input queue */
00160 #define MAX_INPUT 255 /* size of the type-ahead buffer */
00161 #define NAME_MAX DIRSIZ /* # chars in a file name */
00162 #define PATH_MAX 255 /* # chars in a path name */
00163 #define PIPE_BUF 7168 /* # bytes in atomic write to a pipe */
00164 #define STREAM_MAX 20 /* must be the same as FOPEN_MAX in stdio.h */
00165 #define TZNAME_MAX 3 /* maximum bytes in a time zone name is 3 */
00166 #define SSIZE_MAX 32767 /* max defined byte count for read() */
00167
00168 #endif /* _POSIX_SOURCE */
00169
00170 #endif /* _LIMITS_H */

+++++
include(errno.h)
+++++

00200 /* The <errno.h> header defines the numbers of the various errors that can
00201 * occur during program execution. They are visible to user programs and
00202 * should be small positive integers. However, they are also used within
00203 * MINIX, where they must be negative. For example, the READ system call is
00204 * executed internally by calling do_read(). This function returns either a
00205 * (negative) error number or a (positive) number of bytes actually read.
00206 *
00207 * To solve the problem of having the error numbers be negative inside the
00208 * the system and positive outside, the following mechanism is used. All the
00209 * definitions are the form:
00210 *
00211 *     #define EPERM      (_SIGN 1)
00212 *
00213 * If the macro _SYSTEM is defined, then _SIGN is set to "-", otherwise it is
00214 * set to "". Thus when compiling the operating system, the macro _SYSTEM
00215 * will be defined, setting EPERM to (- 1), whereas when when this
00216 * file is included in an ordinary user program, EPERM has the value ( 1).
00217 */
00218
00219 #ifndef _ERRNO_H           /* check if <errno.h> is already included */
00220 #define _ERRNO_H          /* it is not included; note that fact */
00221
00222 /* Now define _SIGN as "" or "--" depending on _SYSTEM. */
00223 #ifdef _SYSTEM
00224 # define _SIGN -
00225 # define OK 0
00226 #else
00227 # define _SIGN
00228 #endif
00229
00230 extern int errno;          /* place where the error numbers go */
00231
00232 /* Here are the numerical values of the error numbers. */
00233 #define _NERROR 70 /* number of errors */
00234

```

```

00235 #define EGENERIC      (_SIGN 99) /* generic error */
00236 #define EPERM        (_SIGN 1)  /* operation not permitted */
00237 #define ENOENT       (_SIGN 2)  /* no such file or directory */
00238 #define ESRCH        (_SIGN 3)  /* no such process */
00239 #define EINTR         (_SIGN 4)  /* interrupted function call */
00240 #define EIO          (_SIGN 5)  /* input/output error */
00241 #define ENXIO        (_SIGN 6)  /* no such device or address */
00242 #define E2BIG         (_SIGN 7)  /* arg list too long */
00243 #define ENOEXEC       (_SIGN 8)  /* exec format error */
00244 #define EBADF         (_SIGN 9)  /* bad file descriptor */
00245 #define ECHILD        (_SIGN 10) /* no child process */
00246 #define EAAGAIN       (_SIGN 11) /* resource temporarily unavailable */
00247 #define ENOMEM        (_SIGN 12) /* not enough space */
00248 #define EACCES        (_SIGN 13) /* permission denied */
00249 #defineEFAULT       (_SIGN 14) /* bad address */
00250 #define ENOTBLK        (_SIGN 15) /* Extension: not a block special file */
00251 #define EBUSY         (_SIGN 16) /* resource busy */
00252 #define EEXIST        (_SIGN 17) /* file exists */
00253 #define EXDEV         (_SIGN 18) /* improper link */
00254 #define ENODEV        (_SIGN 19) /* no such device */
00255 #define ENOTDIR        (_SIGN 20) /* not a directory */
00256 #define EISDIR         (_SIGN 21) /* is a directory */
00257 #define EINVAL        (_SIGN 22) /* invalid argument */
00258 #define ENFILE         (_SIGN 23) /* too many open files in system */
00259 #define EMFILE         (_SIGN 24) /* too many open files */
00260 #define ENOTTY         (_SIGN 25) /* inappropriate I/O control operation */
00261 #define ETXTBSY        (_SIGN 26) /* no longer used */
00262 #define EFBIG          (_SIGN 27) /* file too large */
00263 #define ENOSPC         (_SIGN 28) /* no space left on device */
00264 #define ESPIPE         (_SIGN 29) /* invalid seek */
00265 #define EROFS          (_SIGN 30) /* read-only file system */
00266 #define EMLINK         (_SIGN 31) /* too many links */
00267 #define EPIPE          (_SIGN 32) /* broken pipe */
00268 #define EDOM           (_SIGN 33) /* domain error (from ANSI C std) */
00269 #define ERANGE          (_SIGN 34) /* result too large (from ANSI C std) */
00270 #define EDEADLK         (_SIGN 35) /* resource deadlock avoided */
00271 #define ENAMETOOLONG   (_SIGN 36) /* file name too long */
00272 #define ENOLCK          (_SIGN 37) /* no locks available */
00273 #define ENOSYS          (_SIGN 38) /* function not implemented */
00274 #define ENOTEMPTY       (_SIGN 39) /* directory not empty */
00275
00276 /* The following errors relate to networking. */
00277 #define EPACKSIZE       (_SIGN 50) /* invalid packet size for some protocol */
00278 #define EOOUTOFBUFS    (_SIGN 51) /* not enough buffers left */
00279 #define EBADIOCTL       (_SIGN 52) /* illegal ioctl for device */
00280 #define EBADMODE        (_SIGN 53) /* badmode in ioctl */
00281 #define EWOULDBLOCK     (_SIGN 54)
00282 #define EBADDEST        (_SIGN 55) /* not a valid destination address */
00283 #define EDSTNOTRCH     (_SIGN 56) /* destination not reachable */
00284 #define EISCONN         (_SIGN 57) /* all ready connected */
00285 #define EADDRINUSE      (_SIGN 58) /* address in use */
00286 #define ECONNREFUSED    (_SIGN 59) /* connection refused */
00287 #define ECONNRESET      (_SIGN 60) /* connection reset */
00288 #define ETIMEDOUT       (_SIGN 61) /* connection timed out */
00289 #define EURG            (_SIGN 62) /* urgent data present */
00290 #define ENOURG          (_SIGN 63) /* no urgent data present */
00291 #define ENOTCONN        (_SIGN 64) /* no connection (yet or anymore) */
00292 #define ESHUTDOWN       (_SIGN 65) /* a write call to a shutdown connection */
00293 #define ENOCONN         (_SIGN 66) /* no such connection */
00294 #define EAFNOSUPPORT    (_SIGN 67) /* address family not supported */
00295 #define EPROTONOSUPPORT (_SIGN 68) /* protocol not supported by AF */
00296 #define EPROTOTYPE      (_SIGN 69) /* Protocol wrong type for socket */
00297 #define EINPROGRESS     (_SIGN 70) /* Operation now in progress */
00298 #define EADDRNOTAVAIL   (_SIGN 71) /* Can't assign requested address */
00299 #define EALREADY        (_SIGN 72) /* Connection already in progress */
00300 #define EMSGSIZE        (_SIGN 73) /* Message too long */
00301

```

```

00302 /* The following are not POSIX errors, but they can still happen.
00303 * All of these are generated by the kernel and relate to message passing.
00304 */
00305 #define ELOCKED      (_SIGN 101) /* can't send message due to deadlock */
00306 #define EBADCALL     (_SIGN 102) /* illegal system call number */
00307 #define EBADSRCDST   (_SIGN 103) /* bad source or destination process */
00308 #define ECALLDENIED  (_SIGN 104) /* no permission for system call */
00309 #define EDEADDREADY  (_SIGN 105) /* send destination is not alive */
00310 #define ENOTREADY    (_SIGN 106) /* source or destination is not ready */
00311 #define EBADREQUEST  (_SIGN 107) /* destination cannot handle request */
00312 #define EDONTREPLY   (_SIGN 201) /* pseudo-code: don't send a reply */
00313
00314 #endif /* _ERRNO_H */

+++++
include/unistd.h
+++++

00400 /* The <unistd.h> header contains a few miscellaneous manifest constants. */
00401
00402 #ifndef _UNISTD_H
00403 #define _UNISTD_H
00404
00405 #ifndef _TYPES_H
00406 #include <sys/types.h>
00407 #endif
00408
00409 /* Values used by access(). POSIX Table 2-8. */
00410 #define F_OK          0 /* test if file exists */
00411 #define X_OK          1 /* test if file is executable */
00412 #define W_OK          2 /* test if file is writable */
00413 #define R_OK          4 /* test if file is readable */
00414
00415 /* Values used for whence in lseek(fd, offset, whence). POSIX Table 2-9. */
00416 #define SEEK_SET      0 /* offset is absolute */
00417 #define SEEK_CUR      1 /* offset is relative to current position */
00418 #define SEEK_END      2 /* offset is relative to end of file */
00419
00420 /* This value is required by POSIX Table 2-10. */
00421 #define _POSIX_VERSION 199009L /* which standard is being conformed to */
00422
00423 /* These three definitions are required by POSIX Sec. 8.2.1.2. */
00424 #define STDIN_FILENO   0 /* file descriptor for stdin */
00425 #define STDOUT_FILENO  1 /* file descriptor for stdout */
00426 #define STDERR_FILENO  2 /* file descriptor for stderr */
00427
00428 #ifdef _MINIX
00429 /* How to exit the system or stop a server process. */
00430 #define RBT_HALT      0
00431 #define RBT_REBOOT    1
00432 #define RBT_PANIC    2 /* a server panics */
00433 #define RBT_MONITOR   3 /* let the monitor do this */
00434 #define RBT_RESET    4 /* hard reset the system */
00435 #endif
00436
00437 /* What system info to retrieve with sysgetinfo(). */
00438 #define SI_KINFO      0 /* get kernel info via PM */
00439 #define SI_PROC_ADDR   1 /* address of process table */
00440 #define SI_PROC_TAB    2 /* copy of entire process table */
00441 #define SI_DMAP_TAB   3 /* get device <-> driver mappings */
00442
00443 /* NULL must be defined in <unistd.h> according to POSIX Sec. 2.7.1. */
00444 #define NULL ((void *)0)
00445
00446 /* The following relate to configurable system variables. POSIX Table 4-2. */
00447 #define _SC_ARG_MAX    1
00448 #define _SC_CHILD_MAX  2
00449 #define _SC_CLOCKS_PER_SEC 3

```

```

00450 #define _SC_CLK_TCK      3
00451 #define _SC_NGROUPS_MAX   4
00452 #define _SC_OPEN_MAX      5
00453 #define _SC_JOB_CONTROL    6
00454 #define _SC_SAVED_IDS     7
00455 #define _SC_VERSION       8
00456 #define _SC_STREAM_MAX    9
00457 #define _SC_TZNAME_MAX    10
00458
00459 /* The following relate to configurable pathname variables. POSIX Table 5-2. */
00460 #define _PC_LINK_MAX      1 /* link count */
00461 #define _PC_MAX_CANON     2 /* size of the canonical input queue */
00462 #define _PC_MAX_INPUT      3 /* type-ahead buffer size */
00463 #define _PC_NAME_MAX      4 /* file name size */
00464 #define _PC_PATH_MAX      5 /* pathname size */
00465 #define _PC_PIPE_BUF       6 /* pipe size */
00466 #define _PC_NO_TRUNC      7 /* treatment of long name components */
00467 #define _PC_VDISABLE      8 /* tty disable */
00468 #define _PC_CHOWN_RESTRICTED 9 /* chown restricted or not */
00469
00470 /* POSIX defines several options that may be implemented or not, at the
00471 * implementer's whim. This implementer has made the following choices:
00472 *
00473 * _POSIX_JOB_CONTROL      not defined:      no job control
00474 * _POSIX_SAVED_IDS        not defined:      no saved uid/gid
00475 * _POSIX_NO_TRUNC         defined as -1:    long path names are truncated
00476 * _POSIX_CHOWN_RESTRICTED defined:          you can't give away files
00477 * _POSIX_VDISABLE         defined:          tty functions can be disabled
00478 */
00479 #define _POSIX_NO_TRUNC      (-1)
00480 #define _POSIX_CHOWN_RESTRICTED 1
00481
00482 /* Function Prototypes. */
00483 _PROTOTYPE( void _exit, (int _status) ) ;
00484 _PROTOTYPE( int access, (const char *_path, int _amode) ) ;
00485 _PROTOTYPE( unsigned int alarm, (unsigned int _seconds) ) ;
00486 _PROTOTYPE( int chdir, (const char *_path) ) ;
00487 _PROTOTYPE( int fchdir, (int fd) ) ;
00488 _PROTOTYPE( int chown, (const char *_path, _mnx_Uid_t _owner, _mnx_Gid_t _group) ) ;
00489 _PROTOTYPE( int close, (int _fd) ) ;
00490 _PROTOTYPE( char *ctermid, (char *_s) ) ;
00491 _PROTOTYPE( char *cuserid, (char *_s) ) ;
00492 _PROTOTYPE( int dup, (int _fd) ) ;
00493 _PROTOTYPE( int dup2, (int _fd, int _fd2) ) ;
00494 _PROTOTYPE( int execl, (const char *_path, const char *_arg, ...) ) ;
00495 _PROTOTYPE( int execle, (const char *_path, const char *_arg, ...) ) ;
00496 _PROTOTYPE( int execlp, (const char *_file, const char *_arg, ...) ) ;
00497 _PROTOTYPE( int execv, (const char *_path, char *const _argv[]) ) ;
00498 _PROTOTYPE( int execve, (const char *_path, char *const _argv[],
00499                           char *const _envp[]) ) ;
00500 _PROTOTYPE( int execvp, (const char *_file, char *const _argv[]) ) ;
00501 _PROTOTYPE( pid_t fork, (void) ) ;
00502 _PROTOTYPE( long fpathconf, (int _fd, int _name) ) ;
00503 _PROTOTYPE( char *getcwd, (char *_buf, size_t _size) ) ;
00504 _PROTOTYPE( gid_t getegid, (void) ) ;
00505 _PROTOTYPE( uid_t geteuid, (void) ) ;
00506 _PROTOTYPE( gid_t getgid, (void) ) ;
00507 _PROTOTYPE( int getgroups, (int _gidsetsize, gid_t _grouplist[]) ) ;
00508 _PROTOTYPE( char *getlogin, (void) ) ;
00509 _PROTOTYPE( pid_t getpgrp, (void) ) ;
00510 _PROTOTYPE( pid_t getpid, (void) ) ;
00511 _PROTOTYPE( pid_t getppid, (void) ) ;
00512 _PROTOTYPE( uid_t getuid, (void) ) ;
00513 _PROTOTYPE( int isatty, (int _fd) ) ;
00514 _PROTOTYPE( int link, (const char *_existing, const char *_new) ) ;
00515 _PROTOTYPE( off_t lseek, (int _fd, off_t _offset, int _ whence) ) ;
00516 _PROTOTYPE( long pathconf, (const char *_path, int _name) ) ;

```

```

00517 _PROTOTYPE( int pause, (void) );  

00518 _PROTOTYPE( int pipe, (int _fildes[2]) );  

00519 _PROTOTYPE( ssize_t read, (int _fd, void *_buf, size_t _n) );  

00520 _PROTOTYPE( int rmdir, (const char *_path) );  

00521 _PROTOTYPE( int setgid, (_mnx_Gid_t _gid) );  

00522 _PROTOTYPE( int setpgid, (pid_t _pid, pid_t _pgid) );  

00523 _PROTOTYPE( pid_t setsid, (void) );  

00524 _PROTOTYPE( int setuid, (_mnx_Uid_t _uid) );  

00525 _PROTOTYPE( unsigned int sleep, (unsigned int _seconds) );  

00526 _PROTOTYPE( long sysconf, (int _name) );  

00527 _PROTOTYPE( pid_t tcgetpgrp, (int _fd) );  

00528 _PROTOTYPE( int tcsetpgrp, (int _fd, pid_t _pgrp_id) );  

00529 _PROTOTYPE( char *ttyname, (int _fd) );  

00530 _PROTOTYPE( int unlink, (const char *_path) );  

00531 _PROTOTYPE( ssize_t write, (int _fd, const void *_buf, size_t _n) );  

00532  

00533 /* Open Group Base Specifications Issue 6 (not complete) */  

00534 _PROTOTYPE( int symlink, (const char *_path1, const char *_path2) );  

00535 _PROTOTYPE( int getopt, (int _argc, char **_argv, char *_opts) );  

00536 extern char *optarg;  

00537 extern int optind, opterr, optopt;  

00538 _PROTOTYPE( int usleep, (useconds_t _useconds) );  

00539  

00540 #ifdef _MINIX  

00541 #ifndef _TYPE_H  

00542 #include <minix/type.h>  

00543 #endif  

00544 _PROTOTYPE( int brk, (char *_addr) );  

00545 _PROTOTYPE( int chroot, (const char *_name) );  

00546 _PROTOTYPE( int mknode, (const char *_name, _mnx_Mode_t _mode, Dev_t _addr) );  

00547 _PROTOTYPE( int mknode4, (const char *_name, _mnx_Mode_t _mode, Dev_t _addr,  

00548 long _size) );  

00549 _PROTOTYPE( char *mktemp, (char *_template) );  

00550 _PROTOTYPE( int mount, (char *_spec, char *_name, int _flag) );  

00551 _PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );  

00552 _PROTOTYPE( char *sbrk, (int _incr) );  

00553 _PROTOTYPE( int sync, (void) );  

00554 _PROTOTYPE( int fsync, (int fd) );  

00555 _PROTOTYPE( int umount, (const char *_name) );  

00556 _PROTOTYPE( int reboot, (int _how, ...) );  

00557 _PROTOTYPE( int gethostname, (char *_hostname, size_t _len) );  

00558 _PROTOTYPE( int getdomainname, (char *_domain, size_t _len) );  

00559 _PROTOTYPE( int ttyslot, (void) );  

00560 _PROTOTYPE( int fttyslot, (int _fd) );  

00561 _PROTOTYPE( char *crypt, (const char *_key, const char *_salt) );  

00562 _PROTOTYPE( int getsysinfo, (int who, int what, void *_where) );  

00563 _PROTOTYPE( int getprocnr, (void) );  

00564 _PROTOTYPE( int findproc, (char *proc_name, int *proc_nr) );  

00565 _PROTOTYPE( int allocmem, (phys_bytes size, phys_bytes *base) );  

00566 _PROTOTYPE( int freemem, (phys_bytes size, phys_bytes base) );  

00567 #define DEV_MAP 1  

00568 #define DEV_UNMAP 2  

00569 #define mapdriver(driver, device, style) devctl(DEV_MAP, driver, device, style)  

00570 #define unmapdriver(device) devctl(DEV_UNMAP, 0, device, 0)  

00571 _PROTOTYPE( int devctl, (int ctl_req, int driver, int device, int style));  

00572  

00573 /* For compatibility with other Unix systems */  

00574 _PROTOTYPE( int getpagesize, (void) );  

00575 _PROTOTYPE( int setgroups, (int ngroups, const gid_t *gidset) );  

00576  

00577 #endif  

00578  

00579 _PROTOTYPE( int readlink, (const char *, char *, int));  

00580 _PROTOTYPE( int getopt, (int, char **, char *));  

00581 extern int optind, opterr, optopt;  

00582  

00583 #endif /* _UNISTD_H */

```

```
#####
include/string.h
#####

00600 /* The <string.h> header contains prototypes for the string handling
00601 * functions.
00602 */
00603
00604 #ifndef _STRING_H
00605 #define _STRING_H
00606
00607 #define NULL ((void *)0)
00608
00609 #ifndef _SIZE_T
00610 #define _SIZE_T
00611 typedef unsigned int size_t; /* type returned by sizeof */
00612 #endif /*_SIZE_T */
00613
00614 /* Function Prototypes. */
00615 #ifndef _ANSI_H
00616 #include <ansi.h>
00617 #endif
00618
00619 _PROTOTYPE( void *memchr, (const void *_s, int _c, size_t _n) );
00620 _PROTOTYPE( int memcmp, (const void *_s1, const void *_s2, size_t _n) );
00621 _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n) );
00622 _PROTOTYPE( void *memmove, (void *_s1, const void *_s2, size_t _n) );
00623 _PROTOTYPE( void *memset, (void *_s, int _c, size_t _n) );
00624 _PROTOTYPE( char *strcat, (char *_s1, const char *_s2) );
00625 _PROTOTYPE( char *strchr, (const char *_s, int _c) );
00626 _PROTOTYPE( int strncmp, (const char *_s1, const char *_s2, size_t _n) );
00627 _PROTOTYPE( int strcmp, (const char *_s1, const char *_s2) );
00628 _PROTOTYPE( int strcoll, (const char *_s1, const char *_s2) );
00629 _PROTOTYPE( char *strcpy, (char *_s1, const char *_s2) );
00630 _PROTOTYPE( size_t strcspn, (const char *_s1, const char *_s2) );
00631 _PROTOTYPE( char *strerror, (int _errnum) );
00632 _PROTOTYPE( size_t strlen, (const char *_s) );
00633 _PROTOTYPE( char *strncat, (char *_s1, const char *_s2, size_t _n) );
00634 _PROTOTYPE( char *strncpy, (char *_s1, const char *_s2, size_t _n) );
00635 _PROTOTYPE( char *strupr, (const char *_s1, const char *_s2) );
00636 _PROTOTYPE( char *strlwr, (const char *_s1, const char *_s2) );
00637 _PROTOTYPE( size_t strspn, (const char *_s1, const char *_s2) );
00638 _PROTOTYPE( char *strstr, (const char *_s1, const char *_s2) );
00639 _PROTOTYPE( char *strtok, (char *_s1, const char *_s2) );
00640 _PROTOTYPE( size_t strxfrm, (char *_s1, const char *_s2, size_t _n) );
00641
00642 #ifdef _POSIX_SOURCE
00643 /* Open Group Base Specifications Issue 6 (not complete) */
00644 char *strupr(const char *_s1);
00645 #endif
00646
00647 #ifdef _MINIX
00648 /* For backward compatibility. */
00649 _PROTOTYPE( char *index, (const char *_s, int _charwanted) );
00650 _PROTOTYPE( char *rindex, (const char *_s, int _charwanted) );
00651 _PROTOTYPE( void bcopy, (const void *_src, void *_dst, size_t _length) );
00652 _PROTOTYPE( int bcmp, (const void *_s1, const void *_s2, size_t _length));
00653 _PROTOTYPE( void bzero, (void *_dst, size_t _length) );
00654 _PROTOTYPE( void *memccpy, (char *_dst, const char *_src, int _ucharstop,
00655                                     size_t _size) );
00656
00657 /* Misc. extra functions */
00658 _PROTOTYPE( int strcasecmp, (const char *_s1, const char *_s2) );
00659 _PROTOTYPE( int strncasecmp, (const char *_s1, const char *_s2,
00660                                     size_t _len) );
00661 _PROTOTYPE( size_t strnlen, (const char *_s, size_t _n) );
00662 #endif
```

```

00663
00664 #endif /* _STRING_H */

+++++
           include/signal.h
+++++

00700 /* The <signal.h> header defines all the ANSI and POSIX signals.
00701 * MINIX supports all the signals required by POSIX. They are defined below.
00702 * Some additional signals are also supported.
00703 */
00704
00705 #ifndef _SIGNAL_H
00706 #define _SIGNAL_H
00707
00708 #ifndef _ANSI_H
00709 #include <ansi.h>
00710 #endif
00711 #ifdef _POSIX_SOURCE
00712 #ifndef _TYPES_H
00713 #include <sys/types.h>
00714 #endif
00715 #endif
00716
00717 /* Here are types that are closely associated with signal handling. */
00718 typedef int sig_atomic_t;
00719
00720 #ifdef _POSIX_SOURCE
00721 #ifndef _SIGSET_T
00722 #define _SIGSET_T
00723 typedef unsigned long sigset_t;
00724 #endif
00725 #endif
00726
00727 #define _NSIG          20 /* number of signals used */
00728
00729 #define SIGHUP          1 /* hangup */
00730 #define SIGINT          2 /* interrupt (DEL) */
00731 #define SIGQUIT          3 /* quit (ASCII FS) */
00732 #define SIGILL          4 /* illegal instruction */
00733 #define SIGTRAP          5 /* trace trap (not reset when caught) */
00734 #define SIGABRT          6 /* IOT instruction */
00735 #define SIGIOT          6 /* SIGABRT for people who speak PDP-11 */
00736 #define SIGUNUSED        7 /* spare code */
00737 #define SIGFPE          8 /* floating point exception */
00738 #define SIGKILL          9 /* kill (cannot be caught or ignored) */
00739 #define SIGUSR1          10 /* user defined signal # 1 */
00740 #define SIGSEGV          11 /* segmentation violation */
00741 #define SIGUSR2          12 /* user defined signal # 2 */
00742 #define SIGPIPE          13 /* write on a pipe with no one to read it */
00743 #define SIGALRM          14 /* alarm clock */
00744 #define SIGTERM          15 /* software termination signal from kill */
00745 #define SIGCHLD          17 /* child process terminated or stopped */
00746
00747 #define SIGEMT          7 /* obsolete */
00748 #define SIGBUS          10 /* obsolete */
00749
00750 /* MINIX specific signals. These signals are not used by user processes,
00751 * but meant to inform system processes, like the PM, about system events.
00752 */
00753 #define SIGKMESS         18 /* new kernel message */
00754 #define SIGKSIG          19 /* kernel signal pending */
00755 #define SIGKSTOP         20 /* kernel shutting down */
00756
00757 /* POSIX requires the following signals to be defined, even if they are
00758 * not supported. Here are the definitions, but they are not supported.

```

```

00759  */
00760 #define SIGCONT      18    /* continue if stopped */
00761 #define SIGSTOP       19    /* stop signal */
00762 #define SIGTSTP      20    /* interactive stop signal */
00763 #define SIGTTIN      21    /* background process wants to read */
00764 #define SIGTTOU      22    /* background process wants to write */
00765
00766 /* The sighandler_t type is not allowed unless _POSIX_SOURCE is defined. */
00767 typedef void __PROTOTYPE( (*__sighandler_t), (int) );
00768
00769 /* Macros used as function pointers. */
00770 #define SIG_ERR      ((__sighandler_t) -1)      /* error return */
00771 #define SIG_DFL       ((__sighandler_t) 0)      /* default signal handling */
00772 #define SIG_IGN       ((__sighandler_t) 1)      /* ignore signal */
00773 #define SIG_HOLD      ((__sighandler_t) 2)      /* block signal */
00774 #define SIG_CATCH     ((__sighandler_t) 3)      /* catch signal */
00775 #define SIG_MESS      ((__sighandler_t) 4)      /* pass as message (MINIX) */
00776
00777 #ifdef _POSIX_SOURCE
00778 struct sigaction {
00779     __sighandler_t sa_handler;   /* SIG_DFL, SIG_IGN, or pointer to function */
00780     sigset_t sa_mask;          /* signals to be blocked during handler */
00781     int sa_flags;              /* special flags */
00782 };
00783
00784 /* Fields for sa_flags. */
00785 #define SA_ONSTACK    0x0001  /* deliver signal on alternate stack */
00786 #define SA_RESETHAND  0x0002  /* reset signal handler when signal caught */
00787 #define SA_NODEFER    0x0004  /* don't block signal while catching it */
00788 #define SA_RESTART    0x0008  /* automatic system call restart */
00789 #define SA_SIGINFO    0x0010  /* extended signal handling */
00790 #define SA_NOCLDWAIT  0x0020  /* don't create zombies */
00791 #define SA_NOCLDSTOP  0x0040  /* don't receive SIGCHLD when child stops */
00792
00793 /* POSIX requires these values for use with sigprocmask(2). */
00794 #define SIG_BLOCK     0      /* for blocking signals */
00795 #define SIG_UNBLOCK   1      /* for unblocking signals */
00796 #define SIG_SETMASK   2      /* for setting the signal mask */
00797 #define SIG_INQUIRE   4      /* for internal use only */
00798 #endif /* _POSIX_SOURCE */
00799
00800 /* POSIX and ANSI function prototypes. */
00801 __PROTOTYPE( int raise, (int _sig) ) ;
00802 __PROTOTYPE( __sighandler_t signal, (int _sig, __sighandler_t _func) ) ;
00803
00804 #ifdef _POSIX_SOURCE
00805 __PROTOTYPE( int kill, (pid_t _pid, int _sig) ) ;
00806 __PROTOTYPE( int sigaction,
00807     (int _sig, const struct sigaction *_act, struct sigaction *_oact) ) ;
00808 __PROTOTYPE( int sigaddset, (sigset_t *_set, int _sig) ) ;
00809 __PROTOTYPE( int sigdelset, (sigset_t *_set, int _sig) ) ;
00810 __PROTOTYPE( int sigemptyset, (sigset_t *_set) ) ;
00811 __PROTOTYPE( int sigfillset, (sigset_t *_set) ) ;
00812 __PROTOTYPE( int sigismember, (const sigset_t *_set, int _sig) ) ;
00813 __PROTOTYPE( int sigpending, (sigset_t *_set) ) ;
00814 __PROTOTYPE( int sigprocmask,
00815     (int _how, const sigset_t *_set, sigset_t *_oset) ) ;
00816 __PROTOTYPE( int sigsuspend, (const sigset_t *_sigmask) ) ;
00817 #endif
00818
00819 #endif /* _SIGNAL_H */
+
+=====
+===== include/fcntl.h
+=====+
00900 /* The <fcntl.h> header is needed by the open() and fcntl() system calls,
```

```

00901 * which have a variety of parameters and flags. They are described here.
00902 * The formats of the calls to each of these are:
00903 *
00904 *      open(path, oflag [,mode])      open a file
00905 *      fcntl(fd, cmd [,arg])       get or set file attributes
00906 *
00907 */
00908
00909 #ifndef _FCNTL_H
00910 #define _FCNTL_H
00911
00912 #ifndef _TYPES_H
00913 #include <sys/types.h>
00914 #endif
00915
00916 /* These values are used for cmd in fcntl(). POSIX Table 6-1. */
00917 #define F_DUPFD          0 /* duplicate file descriptor */
00918 #define F_GETFD          1 /* get file descriptor flags */
00919 #define F_SETFD          2 /* set file descriptor flags */
00920 #define F_GETFL          3 /* get file status flags */
00921 #define F_SETFL          4 /* set file status flags */
00922 #define F_GETLK           5 /* get record locking information */
00923 #define F_SETLK           6 /* set record locking information */
00924 #define F_SETLKW          7 /* set record locking info; wait if blocked */
00925
00926 /* File descriptor flags used for fcntl(). POSIX Table 6-2. */
00927 #define FD_CLOEXEC        1 /* close on exec flag for third arg of fcntl */
00928
00929 /* L_type values for record locking with fcntl(). POSIX Table 6-3. */
00930 #define F_RDLCK           1 /* shared or read lock */
00931 #define F_WRLCK           2 /* exclusive or write lock */
00932 #define F_UNLCK           3 /* unlock */
00933
00934 /* Oflag values for open(). POSIX Table 6-4. */
00935 #define O_CREAT            00100 /* creat file if it doesn't exist */
00936 #define O_EXCL             00200 /* exclusive use flag */
00937 #define O_NOCTTY           00400 /* do not assign a controlling terminal */
00938 #define O_TRUNC            01000 /* truncate flag */
00939
00940 /* File status flags for open() and fcntl(). POSIX Table 6-5. */
00941 #define O_APPEND           02000 /* set append mode */
00942 #define O_NONBLOCK          04000 /* no delay */
00943
00944 /* File access modes for open() and fcntl(). POSIX Table 6-6. */
00945 #define O_RDONLY            0 /* open(name, O_RDONLY) opens read only */
00946 #define O_WRONLY             1 /* open(name, O_WRONLY) opens write only */
00947 #define O_RDWR              2 /* open(name, O_RDWR) opens read/write */
00948
00949 /* Mask for use with file access modes. POSIX Table 6-7. */
00950 #define O_ACCMODE           03 /* mask for file access modes */
00951
00952 /* Struct used for locking. POSIX Table 6-8. */
00953 struct flock {
00954     short l_type;           /* type: F_RDLCK, F_WRLCK, or F_UNLCK */
00955     short l_whence;         /* flag for starting offset */
00956     off_t l_start;          /* relative offset in bytes */
00957     off_t l_len;            /* size; if 0, then until EOF */
00958     pid_t l_pid;            /* process id of the locks' owner */
00959 };
00960
00961 /* Function Prototypes. */
00962 _PROTOTYPE( int creat, (const char *_path, _mnx_Mode_t _mode) );
00963 _PROTOTYPE( int fcntl, (int _filedes, int _cmd, ...) );
00964 _PROTOTYPE( int open,  (const char *_path, int _oflag, ...) );
00965
00966 #endif /* _FCNTL_H */

```

```
#####
include/termios.h
#####

01000 /* The <termios.h> header is used for controlling tty modes. */
01001
01002 #ifndef _TERMIOS_H
01003 #define _TERMIOS_H
01004
01005 typedef unsigned short tcflag_t;
01006 typedef unsigned char cc_t;
01007 typedef unsigned int speed_t;
01008
01009 #define NCCS           20 /* size of cc_c array, some extra space
01010          * for extensions. */
01011
01012 /* Primary terminal control structure. POSIX Table 7-1. */
01013 struct termios {
01014     tcflag_t c_iflag;           /* input modes */
01015     tcflag_t c_oflag;           /* output modes */
01016     tcflag_t c_cflag;           /* control modes */
01017     tcflag_t c_lflag;           /* local modes */
01018     speed_t  c_ispeed;          /* input speed */
01019     speed_t  c_ospeed;          /* output speed */
01020     cc_t     c_cc[NCCS];         /* control characters */
01021 };
01022
01023 /* Values for termios c_iflag bit map. POSIX Table 7-2. */
01024 #define BRKINT      0x0001 /* signal interrupt on break */
01025 #define ICRNL       0x0002 /* map CR to NL on input */
01026 #define IGNBRK     0x0004 /* ignore break */
01027 #define IGNCR       0x0008 /* ignore CR */
01028 #define IGNPAR     0x0010 /* ignore characters with parity errors */
01029 #define INLCR       0x0020 /* map NL to CR on input */
01030 #define INPCK       0x0040 /* enable input parity check */
01031 #define ISTRIP      0x0080 /* mask off 8th bit */
01032 #define IXOFF       0x0100 /* enable start/stop input control */
01033 #define IXLON       0x0200 /* enable start/stop output control */
01034 #define PARMRK      0x0400 /* mark parity errors in the input queue */
01035
01036 /* Values for termios c_oflag bit map. POSIX Sec. 7.1.2.3. */
01037 #define OPOST        0x0001 /* perform output processing */
01038
01039 /* Values for termios c_cflag bit map. POSIX Table 7-3. */
01040 #define CLOCAL      0x0001 /* ignore modem status lines */
01041 #define CREAD       0x0002 /* enable receiver */
01042 #define CSIZE       0x000C /* number of bits per character */
01043 #define          CS5  0x0000 /* if CSIZE is CS5, characters are 5 bits */
01044 #define          CS6  0x0004 /* if CSIZE is CS6, characters are 6 bits */
01045 #define          CS7  0x0008 /* if CSIZE is CS7, characters are 7 bits */
01046 #define          CS8  0x000C /* if CSIZE is CS8, characters are 8 bits */
01047 #define CSTOPB     0x0010 /* send 2 stop bits if set, else 1 */
01048 #define HUPCL       0x0020 /* hang up on last close */
01049 #define PARENB     0x0040 /* enable parity on output */
01050 #define PARODD      0x0080 /* use odd parity if set, else even */
01051
01052 /* Values for termios c_lflag bit map. POSIX Table 7-4. */
01053 #define ECHO        0x0001 /* enable echoing of input characters */
01054 #define ECHOE       0x0002 /* echo ERASE as backspace */
01055 #define ECHOK       0x0004 /* echo KILL */
01056 #define ECHONL      0x0008 /* echo NL */
01057 #define ICANON      0x0010 /* canonical input (erase and kill enabled) */
01058 #define IEXTEN      0x0020 /* enable extended functions */
01059 #define ISIG        0x0040 /* enable signals */
01060 #define NOFLSH      0x0080 /* disable flush after interrupt or quit */
01061 #define TOSTOP      0x0100 /* send SIGTTOU (job control, not implemented*)
01062
```

```

01063 /* Indices into c_cc array. Default values in parentheses. POSIX Table 7-5. */
01064 #define VEOF          0 /* cc_c[VEOF] = EOF char (^D) */
01065 #define VEOL          1 /* cc_c[VEOL] = EOL char (undef) */
01066 #define VERASE         2 /* cc_c[VERASE] = ERASE char (^H) */
01067 #define VINTR          3 /* cc_c[VINTR] = INTR char (DEL) */
01068 #define VKILL          4 /* cc_c[VKILL] = KILL char (^U) */
01069 #define VMIN          5 /* cc_c[VMIN] = MIN value for timer */
01070 #define VQUIT          6 /* cc_c[VQUIT] = QUIT char (^V) */
01071 #define VTIME          7 /* cc_c[VTIME] = TIME value for timer */
01072 #define VSUSP          8 /* cc_c[VSUSP] = SUSP (^Z, ignored) */
01073 #define VSTART         9 /* cc_c[VSTART] = START char (^S) */
01074 #define VSTOP          10 /* cc_c[VSTOP] = STOP char (^Q) */

01075
01076 #define _POSIX_VDISABLE (cc_t)0xFF /* You can't even generate this
01077 * character with 'normal' keyboards.
01078 * But some language specific keyboards
01079 * can generate 0xFF. It seems that all
01080 * 256 are used, so cc_t should be a
01081 * short...
01082 */
01083

01084 /* Values for the baud rate settings. POSIX Table 7-6. */
01085 #define B0            0x0000 /* hang up the line */
01086 #define B50           0x1000 /* 50 baud */
01087 #define B75           0x2000 /* 75 baud */
01088 #define B110          0x3000 /* 110 baud */
01089 #define B134          0x4000 /* 134.5 baud */
01090 #define B150          0x5000 /* 150 baud */
01091 #define B200          0x6000 /* 200 baud */
01092 #define B300          0x7000 /* 300 baud */
01093 #define B600          0x8000 /* 600 baud */
01094 #define B1200         0x9000 /* 1200 baud */
01095 #define B1800         0xA000 /* 1800 baud */
01096 #define B2400         0xB000 /* 2400 baud */
01097 #define B4800         0xC000 /* 4800 baud */
01098 #define B9600         0xD000 /* 9600 baud */
01099 #define B19200        0xE000 /* 19200 baud */
01100 #define B38400        0xF000 /* 38400 baud */

01101
01102 /* Optional actions for tcsetattr(). POSIX Sec. 7.2.1.2. */
01103 #define TCSANOW        1 /* changes take effect immediately */
01104 #define TCSADRAIN       2 /* changes take effect after output is done */
01105 #define TCSAFLUSH       3 /* wait for output to finish and flush input */
01106

01107 /* Queue_selector values for tcflush(). POSIX Sec. 7.2.2.2. */
01108 #define TCIFLUSH        1 /* flush accumulated input data */
01109 #define TCOFLUSH        2 /* flush accumulated output data */
01110 #define TCIOFLUSH       3 /* flush accumulated input and output data */

01111
01112 /* Action values for tcflow(). POSIX Sec. 7.2.2.2. */
01113 #define TCOOFF          1 /* suspend output */
01114 #define TCOON           2 /* restart suspended output */
01115 #define TCIOFF          3 /* transmit a STOP character on the line */
01116 #define TCION           4 /* transmit a START character on the line */

01117
01118 /* Function Prototypes. */
01119 #ifndef _ANSI_H
01120 #include <ansi.h>
01121 #endif
01122
01123 _PROTOTYPE( int tcsendbreak, (int _filedes, int _duration) );
01124 _PROTOTYPE( int tcdrain, (int _filedes) );
01125 _PROTOTYPE( int tcflush, (int _filedes, int _queue_selector) );
01126 _PROTOTYPE( int tcflow, (int _filedes, int _action) );
01127 _PROTOTYPE( speed_t cfgetispeed, (const struct termios *_termios_p) );
01128 _PROTOTYPE( speed_t cfgetospeed, (const struct termios *_termios_p) );
01129 _PROTOTYPE( int cfsetispeed, (struct termios *_termios_p, speed_t _speed) );

```

```

01130 _PROTOTYPE( int cfsetospeed, (struct termios *_termios_p, speed_t _speed) );
01131 _PROTOTYPE( int tcgetattr, (int _filedes, struct termios *_termios_p) );
01132 _PROTOTYPE( int tcsetattr, \
01133     (int _filedes, int _opt_actions, const struct termios *_termios_p) );
01134
01135 #define cfgetispeed(termios_p)          ((termios_p)->c_ispeed)
01136 #define cfgetospeed(termios_p)          ((termios_p)->c_ospeed)
01137 #define cfsetispeed(termios_p, speed)   ((termios_p)->c_ispeed = (speed), 0)
01138 #define cfsetospeed(termios_p, speed)   ((termios_p)->c_ospeed = (speed), 0)
01139
01140 #ifdef _MINIX
01141 /* Here are the local extensions to the POSIX standard for Minix. Posix
01142 * conforming programs are not able to access these, and therefore they are
01143 * only defined when a Minix program is compiled.
01144 */
01145
01146 /* Extensions to the termios c_iflag bit map. */
01147 #define IXANY      0x0800 /* allow any key to continue output */
01148
01149 /* Extensions to the termios c_oflag bit map. They are only active iff
01150 * OPOST is enabled. */
01151 #define ONLCR      0x0002 /* Map NL to CR-NL on output */
01152 #define XTABS      0x0004 /* Expand tabs to spaces */
01153 #define ONOEOT     0x0008 /* discard EOT's (^D) on output */
01154
01155 /* Extensions to the termios c_lflag bit map. */
01156 #define LFLUSHO    0x0200 /* Flush output. */
01157
01158 /* Extensions to the c_cc array. */
01159 #define VREPRINT    11 /* cc_c[VREPRINT] (^R) */
01160 #define VLNEXT      12 /* cc_c[VLNEXT] (^V) */
01161 #define VDISCARD    13 /* cc_c[VDISCARD] (^O) */
01162
01163 /* Extensions to baud rate settings. */
01164 #define B57600      0x0100 /* 57600 baud */
01165 #define B115200     0x0200 /* 115200 baud */
01166
01167 /* These are the default settings used by the kernel and by 'stty sane' */
01168
01169 #define TCTRL_DEF    (CREAD | CS8 | HUPCL)
01170 #define TINPUT_DEF    (BRKINT | ICRNL | ION | IXANY)
01171 #define TOUTPUT_DEF   (OPOST | ONLCR)
01172 #define TLOCAL_DEF    (ISIG | IEXTEN | ICANON | ECHO | ECHOE)
01173 #define TSPEED_DEF    B9600
01174
01175 #define TEOF_DEF      '\4' /* ^D */
01176 #define TEOL_DEF      _POSIX_VDISABLE
01177 #define TERASE_DEF    '\10' /* ^H */
01178 #define TINTR_DEF     '\3' /* ^C */
01179 #define TKILL_DEF     '\25' /* ^U */
01180 #define TMIN_DEF      1
01181 #define TQUIT_DEF     '\34' /* ^\ */
01182 #define TSTART_DEF    '\21' /* ^Q */
01183 #define TSTOP_DEF     '\23' /* ^S */
01184 #define TSUSP_DEF     '\32' /* ^Z */
01185 #define TTIME_DEF     0
01186 #define TREPRINT_DEF  '\22' /* ^R */
01187 #define TLNEXT_DEF    '\26' /* ^V */
01188 #define TDISCARD_DEF  '\17' /* ^O */
01189
01190 /* Window size. This information is stored in the TTY driver but not used.
01191 * This can be used for screen based applications in a window environment.
01192 * The ioctl's TIOCGWINSZ and TIOCSWINSZ can be used to get and set this
01193 * information.
01194 */
01195
01196 struct winsize

```



```

01351
01352 /* The following generic timer management functions are available. They
01353 * can be used to operate on the lists of timers. Adding a timer to a list
01354 * automatically takes care of removing it.
01355 */
01356 _PROTOTYPE( clock_t tmrs_clrtimer, (timer_t **tmrs, timer_t *tp, clock_t *new_head)
01357 _PROTOTYPE( void tmrs_exptimers, (timer_t **tmrs, clock_t now, clock_t *new_head)
01358 _PROTOTYPE( clock_t tmrs_settimer, (timer_t **tmrs, timer_t *tp,
01359           clock_t exp_time, tmr_func_t watchdog, clock_t *new_head)
01360
01361 #endif /* _TIMERS_H */
01362

+++++-----+
#include <sys/types.h>
+++++-----+

01400 /* The <sys/types.h> header contains important data type definitions.
01401 * It is considered good programming practice to use these definitions,
01402 * instead of the underlying base type. By convention, all type names end
01403 * with _t.
01404 */
01405
01406 #ifndef _TYPES_H
01407 #define _TYPES_H
01408
01409 #ifndef _ANSI_H
01410 #include <ansi.h>
01411 #endif
01412
01413 /* The type size_t holds all results of the sizeof operator. At first glance,
01414 * it seems obvious that it should be an unsigned int, but this is not always
01415 * the case. For example, MINIX-ST (68000) has 32-bit pointers and 16-bit
01416 * integers. When one asks for the size of a 70K struct or array, the result
01417 * requires 17 bits to express, so size_t must be a long type. The type
01418 * ssize_t is the signed version of size_t.
01419 */
01420 #ifndef _SIZE_T
01421 #define _SIZE_T
01422 typedef unsigned int size_t;
01423 #endif
01424
01425 #ifndef _SSIZE_T
01426 #define _SSIZE_T
01427 typedef int ssize_t;
01428 #endif
01429
01430 #ifndef _TIME_T
01431 #define _TIME_T
01432 typedef long time_t;           /* time in sec since 1 Jan 1970 0000 GMT */
01433 #endif
01434
01435 #ifndef _CLOCK_T
01436 #define _CLOCK_T
01437 typedef long clock_t;          /* unit for system accounting */
01438 #endif
01439
01440 #ifndef _SIGSET_T
01441 #define _SIGSET_T
01442 typedef unsigned long sigset_t;
01443 #endif
01444

```

```

01445 /* Open Group Base Specifications Issue 6 (not complete) */
01446 typedef long useconds_t;           /* Time in microseconds */
01447
01448 /* Types used in disk, inode, etc. data structures. */
01449 typedef short      dev_t;        /* holds (major|minor) device pair */
01450 typedef char       gid_t;        /* group id */
01451 typedef unsigned long ino_t;     /* i-node number (V3 filesystem) */
01452 typedef unsigned short mode_t;   /* file type and permissions bits */
01453 typedef short      nlink_t;     /* number of links to a file */
01454 typedef unsigned long off_t;    /* offset within a file */
01455 typedef int         pid_t;        /* process id (must be signed) */
01456 typedef short      uid_t;        /* user id */
01457 typedef unsigned long zone_t;   /* zone number */
01458 typedef unsigned long block_t;  /* block number */
01459 typedef unsigned long bit_t;    /* bit number in a bit map */
01460 typedef unsigned short zone1_t; /* zone number for V1 file systems */
01461 typedef unsigned short bchunk_t; /* collection of bits in a bitmap */
01462
01463 typedef unsigned char   u8_t;     /* 8 bit type */
01464 typedef unsigned short  u16_t;    /* 16 bit type */
01465 typedef unsigned long   u32_t;    /* 32 bit type */
01466
01467 typedef char          i8_t;     /* 8 bit signed type */
01468 typedef short         i16_t;    /* 16 bit signed type */
01469 typedef long          i32_t;    /* 32 bit signed type */
01470
01471 typedef struct { u32_t _[2]; } u64_t;
01472
01473 /* The following types are needed because MINIX uses K&R style function
01474 * definitions (for maximum portability). When a short, such as dev_t, is
01475 * passed to a function with a K&R definition, the compiler automatically
01476 * promotes it to an int. The prototype must contain an int as the parameter,
01477 * not a short, because an int is what an old-style function definition
01478 * expects. Thus using dev_t in a prototype would be incorrect. It would be
01479 * sufficient to just use int instead of dev_t in the prototypes, but Dev_t
01480 * is clearer.
01481 */
01482 typedef int          Dev_t;
01483 typedef int          _mnx_Gid_t;
01484 typedef int          Nlink_t;
01485 typedef int          _mnx_Uid_t;
01486 typedef int          U8_t;
01487 typedef unsigned long U32_t;
01488 typedef int          I8_t;
01489 typedef int          I16_t;
01490 typedef long         I32_t;
01491
01492 /* ANSI C makes writing down the promotion of unsigned types very messy. When
01493 * sizeof(short) == sizeof(int), there is no promotion, so the type stays
01494 * unsigned. When the compiler is not ANSI, there is usually no loss of
01495 * unsignedness, and there are usually no prototypes so the promoted type
01496 * doesn't matter. The use of types like Ino_t is an attempt to use ints
01497 * (which are not promoted) while providing information to the reader.
01498 */
01499
01500 typedef unsigned long Ino_t;
01501
01502 #if _EM_WSIZE == 2
01503 /*typedef unsigned int    Ino_t; Ino_t is now 32 bits */
01504 typedef unsigned int    Zone1_t;
01505 typedef unsigned int    Bchunk_t;
01506 typedef unsigned int    U16_t;
01507 typedef unsigned int    _mnx_Mode_t;
01508
01509 #else /* _EM_WSIZE == 4, or _EM_WSIZE undefined */
01510 /*typedef int            Ino_t; Ino_t is now 32 bits */
01511 typedef int            Zone1_t;

```

```

01512     typedef int          Bitchunk_t;
01513     typedef int          U16_t;
01514     typedef int          _mnx_Mode_t;
01515
01516 #endif /* _EM_WSIZE == 2, etc */
01517
01518 /* Signal handler type, e.g. SIG_IGN */
01519     typedef void _PROTOTYPE( (*sighandler_t), (int) );
01520
01521 /* Compatibility with other systems */
01522     typedef unsigned char   u_char;
01523     typedef unsigned short  u_short;
01524     typedef unsigned int    u_int;
01525     typedef unsigned long   u_long;
01526     typedef char           *caddr_t;
01527
01528 #endif /* _TYPES_H */

+
+-----+
include/sys/sigcontext.h
+-----+

01600 #ifndef _SIGCONTEXT_H
01601 #define _SIGCONTEXT_H
01602
01603 /* The sigcontext structure is used by the sigreturn(2) system call.
01604 * sigreturn() is seldom called by user programs, but it is used internally
01605 * by the signal catching mechanism.
01606 */
01607
01608 #ifndef _ANSI_H
01609 #include <ansi.h>
01610 #endif
01611
01612 #ifndef _MINIX_SYS_CONFIG_H
01613 #include <minix/sys_config.h>
01614 #endif
01615
01616 #if !defined(_MINIX_CHIP)
01617 #include "error, configuration is not known"
01618 #endif
01619
01620 /* The following structure should match the stackframe_s structure used
01621 * by the kernel's context switching code. Floating point registers should
01622 * be added in a different struct.
01623 */
01624 struct sigregs {
01625     short sr_gs;
01626     short sr_fs;
01627     short sr_es;
01628     short sr_ds;
01629     int sr_di;
01630     int sr_si;
01631     int sr_bp;
01632     int sr_st;           /* stack top -- used in kernel */
01633     int sr_bx;
01634     int sr_dx;
01635     int sr_cx;
01636     int sr_retreg;
01637     int sr_retadr;       /* return address to caller of save -- used
01638                           * in kernel */
01639     int sr_pc;
01640     int sr_cs;
01641     int sr_psw;
01642     int sr_sp;
01643     int sr_ss;

```

```

01644 };
01645
01646 struct sigframe { /* stack frame created for signalled process */
01647     _PROTOTYPE( void (*sf_retadr), (void) );
01648     int sf_signo;
01649     int sf_code;
01650     struct sigcontext *sf_scp;
01651     int sf_fp;
01652     _PROTOTYPE( void (*sf_retadr2), (void) );
01653     struct sigcontext *sf_scpcopy;
01654 };
01655
01656 struct sigcontext {
01657     int sc_flags;           /* sigstack state to restore */
01658     long sc_mask;          /* signal mask to restore */
01659     struct sigregs sc_regs; /* register set to restore */
01660 };
01661
01662 #define sc_gs sc_regs.sr_gs
01663 #define sc_fs sc_regs.sr_fs
01664 #define sc_es sc_regs.sr_es
01665 #define sc_ds sc_regs.sr_ds
01666 #define sc_di sc_regs.sr_di
01667 #define sc_si sc_regs.sr_si
01668 #define sc_fp sc_regs.sr_bp
01669 #define sc_st sc_regs.sr_st           /* stack top -- used in kernel */
01670 #define sc_bx sc_regs.sr_bx
01671 #define sc_dx sc_regs.sr_dx
01672 #define sc_cx sc_regs.sr_cx
01673 #define sc_retreg sc_regs.sr_retreg
01674 #define sc_retadr sc_regs.sr_retadr /* return address to caller of
01675                                         save -- used in kernel */
01676 #define sc_pc sc_regs.sr_pc
01677 #define sc_cs sc_regs.sr_cs
01678 #define sc_psw sc_regs.sr_psw
01679 #define sc_sp sc_regs.sr_sp
01680 #define sc_ss sc_regs.sr_ss
01681
01682 /* Values for sc_flags. Must agree with <minix/jmp_buf.h>. */
01683 #define SC_SIGCONTEXT 2      /* nonzero when signal context is included */
01684 #define SC_NOREGLOCALS 4     /* nonzero when registers are not to be
01685                                         saved and restored */
01686
01687 _PROTOTYPE( int sigreturn, (struct sigcontext *_scp) );
01688
01689 #endif /* _SIGCONTEXT_H */

```

+++++  
 include/sys/stat.h  
 ++++++

```

01700 /* The <sys/stat.h> header defines a struct that is used in the stat() and
01701 * fstat functions. The information in this struct comes from the i-node of
01702 * some file. These calls are the only approved way to inspect i-nodes.
01703 */
01704
01705 #ifndef _STAT_H
01706 #define _STAT_H
01707
01708 #ifndef _TYPES_H
01709 #include <sys/types.h>
01710 #endif
01711
01712 struct stat {
01713     dev_t st_dev;           /* major/minor device number */

```

```

01714     ino_t st_ino;           /* i-node number */
01715     mode_t st_mode;        /* file mode, protection bits, etc. */
01716     short int st_nlink;    /* # links; TEMPORARY HACK: should be nlink_t*/
01717     uid_t st_uid;          /* uid of the file's owner */
01718     short int st_gid;       /* gid; TEMPORARY HACK: should be gid_t */
01719     dev_t st_rdev;
01720     off_t st_size;         /* file size */
01721     time_t st_atime;       /* time of last access */
01722     time_t st_mtime;       /* time of last data modification */
01723     time_t st_ctime;       /* time of last file status change */
01724 };
01725
01726 /* Traditional mask definitions for st_mode. */
01727 /* The ugly casts on only some of the definitions are to avoid suprising sign
01728 * extensions such as S_IFREG != (mode_t) S_IFREG when ints are 32 bits.
01729 */
01730 #define S_IFMT ((mode_t) 0170000)      /* type of file */
01731 #define S_IFLNK ((mode_t) 0120000)     /* symbolic link, not implemented */
01732 #define S_IFREG ((mode_t) 0100000)     /* regular */
01733 #define S_IFBLK 0060000             /* block special */
01734 #define S_IFDIR 0040000             /* directory */
01735 #define S_IFCHR 0020000             /* character special */
01736 #define S_IFIFO 0010000             /* this is a FIFO */
01737 #define S_ISUID 0004000             /* set user id on execution */
01738 #define S_ISGID 0002000             /* set group id on execution */
01739 #define S_ISVTX 01000              /* next is reserved for future use */
01740 #define S_ISVTX 01000              /* save swapped text even after use */
01741
01742 /* POSIX masks for st_mode. */
01743 #define S_IRWXU 00700             /* owner: rwx----- */
01744 #define S_IRUSR 00400             /* owner: r----- */
01745 #define S_IWUSR 00200             /* owner: -w----- */
01746 #define S_IXUSR 00100             /* owner: --x----- */
01747
01748 #define S_IRWXG 00070             /* group: ---rwx--- */
01749 #define S_IRGRP 00040             /* group: ---r---- */
01750 #define S_IWGRP 00020             /* group: ----w--- */
01751 #define S_IXGRP 00010             /* group: -----x-- */
01752
01753 #define S_IRWXO 00007             /* others: -----rwx */
01754 #define S_IROTH 00004             /* others: -----r-- */
01755 #define S_IWOTH 00002             /* others: -----w- */
01756 #define S_IXOTH 00001             /* others: -----x- */
01757
01758 /* The following macros test st_mode (from POSIX Sec. 5.6.1.1). */
01759 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)      /* is a reg file */
01760 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR)      /* is a directory */
01761 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR)      /* is a char spec */
01762 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK)      /* is a block spec */
01763 #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO)      /* is a pipe/FIFO */
01764 #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK)      /* is a sym link */
01765
01766 /* Function Prototypes. */
01767 _PROTOTYPE( int chmod, (const char *_path, _mnx_Mode_t _mode) );
01768 _PROTOTYPE( int fstat, (int _fildes, struct stat *_buf) );
01769 _PROTOTYPE( int mkdir, (const char *_path, _mnx_Mode_t _mode) );
01770 _PROTOTYPE( int mkfifo, (const char *_path, _mnx_Mode_t _mode) );
01771 _PROTOTYPE( int stat, (const char *_path, struct stat *_buf) );
01772 _PROTOTYPE( mode_t umask, (_mnx_Mode_t _cmask) );
01773
01774 /* Open Group Base Specifications Issue 6 (not complete) */
01775 _PROTOTYPE( int lstat, (const char *_path, struct stat *_buf) );
01776
01777 #endif /* _STAT_H */

```

```
+++++
include/sys/dir.h
+++++
01800 /* The <dir.h> header gives the layout of a directory. */
01801 #ifndef _DIR_H
01802 #define _DIR_H
01803
01804
01805 #include <sys/types.h>
01806
01807 #define DIRBLKSIZ      512    /* size of directory block */
01808
01809 #ifndef DIRSIZ
01810 #define DIRSIZ 60
01811 #endif
01812
01813 struct direct {
01814     ino_t d_ino;
01815     char d_name[DIRSIZ];
01816 };
01817
01818 #endif /* _DIR_H */

+++++
include/sys/wait.h
+++++
01900 /* The <sys/wait.h> header contains macros related to wait(). The value
01901 * returned by wait() and waitpid() depends on whether the process
01902 * terminated by an exit() call, was killed by a signal, or was stopped
01903 * due to job control, as follows:
01904 *
01905 *          High byte   Low byte
01906 *          +-----+
01907 *          | status | 0 |
01908 *          +-----+
01909 *          killed by signal   | 0 | signal |
01910 *          +-----+
01911 *          stopped (job control) | signal | 0177 |
01912 *          +-----+
01913 */
01914
01915 #ifndef _WAIT_H
01916 #define _WAIT_H
01917
01918 #ifndef _TYPES_H
01919 #include <sys/types.h>
01920 #endif
01921
01922 #define _LOW(v)        ((v) & 0377)
01923 #define _HIGH(v)       (((v) >> 8) & 0377)
01924
01925 #define WNOHANG        1      /* do not wait for child to exit */
01926 #define WUNTRACED      2      /* for job control; not implemented */
01927
01928 #define WIFEXITED(s)   (_LOW(s) == 0)           /* normal exit */
01929 #define WEXITSTATUS(s) (_HIGH(s))            /* exit status */
01930 #define WTERMSIG(s)    (_LOW(s) & 0177)         /* sig value */
01931 #define WIFSIGNALED(s) (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) /* signaled */
01932 #define WIFSTOPPED(s)  (_LOW(s) == 0177)         /* stopped */
01933 #define WSTOPSIG(s)   (_HIGH(s) & 0377)         /* stop signal */
01934
01935 /* Function Prototypes. */
01936 _PROTOTYPE( pid_t wait, (int *_stat_loc) );
01937 _PROTOTYPE( pid_t waitpid, (pid_t _pid, int *_stat_loc, int _options) );
```

```

01938
01939 #endif /* _WAIT_H */

+++++
include/sys/ioctl.h
+++++

02000 /*      sys/ioctl.h - All ioctl() command codes.          Author: Kees J. Bot
02001 *
02002 *
02003 * This header file includes all other ioctl command code headers.
02004 */
02005
02006 #ifndef _S_IOCTL_H
02007 #define _S_IOCTL_H
02008
02009 /* A driver that uses ioctls claims a character for its series of commands.
02010 * For instance: #define TCGETS _IOR('T', 8, struct termios)
02011 * This is a terminal ioctl that uses the character 'T'. The character(s)
02012 * used in each header file are shown in the comment following.
02013 */
02014
02015 #include <sys/ioc_tty.h>      /* 'T' 't' 'k' */
02016 #include <sys/ioc_disk.h>      /* 'd' */
02017 #include <sys/ioc_memory.h>    /* 'm' */
02018 #include <sys/ioc_cmos.h>      /* 'c' */
02019
02020 #endif /* _S_IOCTL_H */

+++++
include/sys/ioc_disk.h
+++++

02100 /*      sys/ioc_disk.h - Disk ioctl() command codes.        Author: Kees J. Bot
02101 *
02102 *
02103 */
02104
02105 #ifndef _S_I_DISK_H
02106 #define _S_I_DISK_H
02107
02108 #include <minix/ioc.h>
02109
02110 #define DIOCSETP      _IOW('d', 3, struct partition)
02111 #define DIOCGETP      _IOR('d', 4, struct partition)
02112 #define DIOCEJECT     _IO ('d', 5)
02113 #define DIOCTIMEOUT   _IOW('d', 6, int)
02114 #define DIOPENCNT     _IOR('d', 7, int)
02115
02116 #endif /* _S_I_DISK_H */

+++++
include/minix/ioc.h
+++++

02200 /*      minix/ioc.h - Ioctl helper definitions.          Author: Kees J. Bot
02201 *
02202 *
02203 * This file is included by every header file that defines ioctl codes.
02204 */

```

```

02205
02206 #ifndef _M_IOCTL_H
02207 #define _M_IOCTL_H
02208
02209 #ifndef _TYPES_H
02210 #include <sys/types.h>
02211 #endif
02212
02213 #if _EM_WSIZE >= 4
02214 /* Iocctl have the command encoded in the low-order word, and the size
02215 * of the parameter in the high-order word. The 3 high bits of the high-
02216 * order word are used to encode the in/out/void status of the parameter.
02217 */
02218 #define _IOCPARM_MASK 0x1FFF
02219 #define _IOC_VOID 0x20000000
02220 #define _IOCTYPE_MASK 0xFFFF
02221 #define _IOC_IN 0x40000000
02222 #define _IOC_OUT 0x80000000
02223 #define _IOC_INOUT (_IOC_IN | _IOC_OUT)
02224
02225 #define _IO(x,y) ((x << 8) | y | _IOC_VOID)
02226 #define _IOR(x,y,t) ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) | \
02227 _IOC_OUT)
02228 #define _IOW(x,y,t) ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) | \
02229 _IOC_IN)
02230 #define _IORW(x,y,t) ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) | \
02231 _IOC_INOUT)
02232 #else
02233 /* No fancy encoding on a 16-bit machine. */
02234
02235 #define _IO(x,y) ((x << 8) | y)
02236 #define _IOR(x,y,t) _IO(x,y)
02237 #define _IOW(x,y,t) _IO(x,y)
02238 #define _IORW(x,y,t) _IO(x,y)
02239 #endif
02240
02241 int ioctl(int _fd, int _request, void *_data);
02242
02243 #endif /* _M_IOCTL_H */

```

```

+++++
#include/minix/config.h
+++++
02300 #ifndef _CONFIG_H
02301 #define _CONFIG_H
02302
02303 /* Minix release and version numbers. */
02304 #define OS_RELEASE "3"
02305 #define OS_VERSION "1.0"
02306
02307 /* This file sets configuration parameters for the MINIX kernel, FS, and PM.
02308 * It is divided up into two main sections. The first section contains
02309 * user-settable parameters. In the second section, various internal system
02310 * parameters are set based on the user-settable parameters.
02311 *
02312 * Parts of config.h have been moved to sys_config.h, which can be included
02313 * by other include files that wish to get at the configuration data, but
02314 * don't want to pollute the users namespace. Some editable values have
02315 * gone there.
02316 *
02317 * This is a modified version of config.h for compiling a small Minix system
02318 * with only the options described in the text, Operating Systems Design and
02319 * Implementation, 3rd edition. See the version of config.h in the full
02320 * source code directory for information on alternatives omitted here.
02321 */

```

```

02322
02323 /* The MACHINE (called _MINIX_MACHINE) setting can be done
02324 * in <minix/machine.h>.
02325 */
02326 #include <minix/sys_config.h>
02327
02328 #define MACHINE _MINIX_MACHINE
02329
02330 #define IBM_PC _MACHINE_IBM_PC
02331 #define SUN_4 _MACHINE_SUN_4
02332 #define SUN_4_60 _MACHINE_SUN_4_60
02333 #define ATARI _MACHINE_ATARI
02334 #define MACINTOSH _MACHINE_MACINTOSH
02335
02336 /* Number of slots in the process table for non-kernel processes. The number
02337 * of system processes defines how many processes with special privileges
02338 * there can be. User processes share the same properties and count for one.
02339 */
02340 * These can be changed in sys_config.h.
02341 */
02342 #define NR_PROCS _NR_PROCS
02343 #define NR_SYS_PROCS _NR_SYS_PROCS
02344
02345 #define NR_BUFS 128
02346 #define NR_BUF_HASH 128
02347
02348 /* Number of controller tasks (/dev/cN device classes). */
02349 #define NR_CTRLRS 2
02350
02351 /* Enable or disable the second level file system cache on the RAM disk. */
02352 #define ENABLE_CACHE2 0
02353
02354 /* Enable or disable swapping processes to disk. */
02355 #define ENABLE_SWAP 0
02356
02357 /* Include or exclude an image of /dev/boot in the boot image.
02358 * Please update the makefile in /usr/src/tools/ as well.
02359 */
02360 #define ENABLE_BOOTDEV 0 /* load image of /dev/boot at boot time */
02361
02362 /* DMA_SECTORS may be increased to speed up DMA based drivers. */
02363 #define DMA_SECTORS 1 /* DMA buffer size (must be >= 1) */
02364
02365 /* Include or exclude backwards compatibility code. */
02366 #define ENABLE_BINCOMPAT 0 /* for binaries using obsolete calls */
02367 #define ENABLE_SRCCOMPAT 0 /* for sources using obsolete calls */
02368
02369 /* Which process should receive diagnostics from the kernel and system?
02370 * Directly sending it to TTY only displays the output. Sending it to the
02371 * log driver will cause the diagnostics to be buffered and displayed.
02372 */
02373 #define OUTPUT_PROC_NR LOG_PROC_NR /* TTY_PROC_NR or LOG_PROC_NR */
02374
02375 /* NR_CONS, NR_RS_LINES, and NR_PTYS determine the number of terminals the
02376 * system can handle.
02377 */
02378 #define NR_CONS 4 /* # system consoles (1 to 8) */
02379 #define NR_RS_LINES 0 /* # rs232 terminals (0 to 4) */
02380 #define NR_PTYS 0 /* # pseudo terminals (0 to 64) */
02381
02382 /*=====
02383 * There are no user-settable parameters after this line
02384 *=====
02385 */ Set the CHIP type based on the machine selected. The symbol CHIP is actually
02386 * indicative of more than just the CPU. For example, machines for which
02387 * CHIP == INTEL are expected to have 8259A interrupt controllers and the
02388 * other properties of IBM PC/XT/AT/386 types machines in general.

```

```

02389 #define INTEL           _CHIP_INTEL /* CHIP type for PC, XT, AT, 386 and clones */
02390 #define M68000          _CHIP_M68000 /* CHIP type for Atari, Amiga, Macintosh */
02391 #define SPARC            _CHIP_SPARC /* CHIP type for SUN-4 (e.g. SPARCstation) */
02392
02393 /* Set the FP_FORMAT type based on the machine selected, either hw or sw */
02394 #define FP_NONE          _FP_NONE /* no floating point support */
02395 #define FP_IEEE           _FP_IEEE /* conform IEEE floating point standard */
02396
02397 /* _MINIX_CHIP is defined in sys_config.h. */
02398 #define CHIP              _MINIX_CHIP
02399
02400 /* _MINIX_FP_FORMAT is defined in sys_config.h. */
02401 #define FP_FORMAT         _MINIX_FP_FORMAT
02402
02403 /* _ASKDEV and _FASTLOAD are defined in sys_config.h. */
02404 #define ASKDEV            _ASKDEV
02405 #define FASTLOAD          _FASTLOAD
02406
02407 #endif /* _CONFIG_H */

+++++
include/minix/sys_config.h
+++++

02500 #ifndef _MINIX_SYS_CONFIG_H
02501 #define _MINIX_SYS_CONFIG_H 1
02502
02503 /* This is a modified sys_config.h for compiling a small Minix system
   * with only the options described in the text, Operating Systems Design and
   * Implementation, 3rd edition. See the sys_config.h in the full
   * source code directory for information on alternatives omitted here.
02507 */
02508
02509 /*=====
02510 *      This section contains user-settable parameters
02511 *=====
02512 #define _MINIX_MACHINE      _MACHINE_IBM_PC
02513
02514 #define _MACHINE_IBM_PC     1 /* any 8088 or 80x86-based system */
02515
02516 /* Word size in bytes (a constant equal to sizeof(int)). */
02517 #if __ACK__ || __GNUC__
02518 #define _WORD_SIZE          _EM_WSIZE
02519 #define _PTR_SIZE           _EM_WSIZE
02520 #endif
02521
02522 #define _NR_PROCS           64
02523 #define _NR_SYS_PROCS       32
02524
02525 /* Set the CHIP type based on the machine selected. The symbol CHIP is actually
   * indicative of more than just the CPU. For example, machines for which
   * CHIP == INTEL are expected to have 8259A interrupt controllers and the
   * other properties of IBM PC/XT/AT/386 types machines in general. */
02529 #define _CHIP_INTEL          1 /* CHIP type for PC, XT, AT, 386 and clones */
02530
02531 /* Set the FP_FORMAT type based on the machine selected, either hw or sw */
02532 #define _FP_NONE            0 /* no floating point support */
02533 #define _FP_IEEE             1 /* conform IEEE floating point standard */
02534
02535 #define _MINIX_CHIP          _CHIP_INTEL
02536
02537 #define _MINIX_FP_FORMAT     _FP_NONE
02538
02539 #ifndef _MINIX_MACHINE

```

```

02540 error "In <minix/sys_config.h> please define _MINIX_MACHINE"
02541 #endif
02542
02543 #ifndef _MINIX_CHIP
02544 error "In <minix/sys_config.h> please define _MINIX_MACHINE to have a legal value"
02545 #endif
02546
02547 #if (_MINIX_MACHINE == 0)
02548 error "_MINIX_MACHINE has incorrect value (0)"
02549 #endif
02550
02551 #endif /* _MINIX_SYS_CONFIG_H */
02552
02553

+++++
include/minix/const.h
++++

02600 /* Copyright (C) 2001 by Prentice-Hall, Inc. See the copyright notice in
02601 * the file /usr/src/LICENSE.
02602 */
02603
02604 #ifndef CHIP
02605 #error CHIP is not defined
02606 #endif
02607
02608 #define EXTERN      extern    /* used in *.h files */
02609 #define PRIVATE     static    /* PRIVATE x limits the scope of x */
02610 #define PUBLIC      /* PUBLIC is the opposite of PRIVATE */
02611 #define FORWARD     static    /* some compilers require this to be 'static'*/
02612
02613 #define TRUE        1        /* used for turning integers into Booleans */
02614 #define FALSE       0        /* used for turning integers into Booleans */
02615
02616 #define HZ          60       /* clock freq (software settable on IBM-PC) */
02617
02618 #define SUPER_USER (uid_t) 0 /* uid_t of superuser */
02619
02620 /* Devices. */
02621 #define MAJOR        8        /* major device = (dev>>MAJOR) & 0377 */
02622 #define MINOR        0        /* minor device = (dev>>MINOR) & 0377 */
02623
02624 #define NULL    ((void *)0) /* null pointer */
02625 #define CPVEC_NR      16       /* max # of entries in a SYS_VCOPY request */
02626 #define CPVVEC_NR     64       /* max # of entries in a SYS_VCOPY request */
02627 #define NR_IOREQS   MIN(NR_BUFS, 64)
02628                                /* maximum number of entries in an iorequest */
02629
02630 /* Message passing constants. */
02631 #define MESS_SIZE (sizeof(message)) /* might need usizeof from FS here */
02632 #define NIL_MESS ((message *) 0) /* null pointer */
02633
02634 /* Memory related constants. */
02635 #define SEGMENT_TYPE 0xFF00 /* bit mask to get segment type */
02636 #define SEGMENT_INDEX 0x0FFF /* bit mask to get segment index */
02637
02638 #define LOCAL_SEG   0x0000 /* flags indicating local memory segment */
02639 #define NR_LOCAL_SEGS 3 /* # local segments per process (fixed) */
02640 #define T           0 /* proc[i].mem_map[T] is for text */
02641 #define D           1 /* proc[i].mem_map[D] is for data */
02642 #define S           2 /* proc[i].mem_map[S] is for stack */
02643
02644 #define REMOTE_SEG   0x0100 /* flags indicating remote memory segment */
02645 #define NR_REMOTE_SEGS 3 /* # remote memory regions (variable) */
02646

```

```

02647 #define BIOS_SEG      0x0200 /* flags indicating BIOS memory segment */
02648 #define NR_BIOS_SEGS    3 /* # BIOS memory regions (variable) */
02649
02650 #define PHYS_SEG       0x0400 /* flag indicating entire physical memory */
02651
02652 /* Labels used to disable code sections for different reasons. */
02653 #define DEAD_CODE        0 /* unused code in normal configuration */
02654 #define FUTURE_CODE       0 /* new code to be activated + tested later */
02655 #define TEMP_CODE         1 /* active code to be removed later */
02656
02657 /* Process name length in the PM process table, including '\0'. */
02658 #define PROC_NAME_LEN     16
02659
02660 /* Miscellaneous */
02661 #define BYTE            0377 /* mask for 8 bits */
02662 #define READING          0 /* copy data to user */
02663 #define WRITING          1 /* copy data from user */
02664 #define NO_NUM           0x8000 /* used as numerical argument to panic() */
02665 #define NIL_PTR          (char *) 0 /* generally useful expression */
02666 #define HAVE_SCATTERED_IO 1 /* scattered I/O is now standard */
02667
02668 /* Macros. */
02669 #define MAX(a, b) ((a) > (b) ? (a) : (b))
02670 #define MIN(a, b) ((a) < (b) ? (a) : (b))
02671
02672 /* Memory is allocated in clicks. */
02673 #if (CHIP == INTEL)
02674 #define CLICK_SIZE        1024 /* unit in which memory is allocated */
02675 #define CLICK_SHIFT       10 /* log2 of CLICK_SIZE */
02676 #endif
02677
02678 #if (CHIP == SPARC) || (CHIP == M68000)
02679 #define CLICK_SIZE        4096 /* unit in which memory is allocated */
02680 #define CLICK_SHIFT       12 /* log2 of CLICK_SIZE */
02681 #endif
02682
02683 /* Click to byte conversions (and vice versa). */
02684 #define HCLICK_SHIFT      4 /* log2 of HCLICK_SIZE */
02685 #define HCLICK_SIZE        16 /* hardware segment conversion magic */
02686 #if CLICK_SIZE >= HCLICK_SIZE
02687 #define click_to_hclick(n) ((n) << (CLICK_SHIFT - HCLICK_SHIFT))
02688 #else
02689 #define click_to_hclick(n) ((n) >> (HCLICK_SHIFT - CLICK_SHIFT))
02690 #endif
02691 #define hclick_to_physb(n) ((phys_bytes) (n) << HCLICK_SHIFT)
02692 #define physb_to_hclick(n) ((n) >> HCLICK_SHIFT)
02693
02694 #define ABS                -999 /* this process means absolute memory */
02695
02696 /* Flag bits for i_mode in the inode. */
02697 #define I_TYPE             0170000 /* this field gives inode type */
02698 #define I_REGULAR          0100000 /* regular file, not dir or special */
02699 #define I_BLOCK_SPECIAL    0060000 /* block special file */
02700 #define I_DIRECTORY         0040000 /* file is a directory */
02701 #define I_CHAR_SPECIAL     0020000 /* character special file */
02702 #define I_NAMED_PIPE        0010000 /* named pipe (FIFO) */
02703 #define I_SET_UID_BIT       0004000 /* set effective uid_t on exec */
02704 #define I_SET_GID_BIT       0002000 /* set effective gid_t on exec */
02705 #define ALL_MODES          0006777 /* all bits for user, group and others */
02706 #define RWX_MODES          0000777 /* mode bits for RWX only */
02707 #define R_BIT               0000004 /* Rx protection bit */
02708 #define W_BIT               0000002 /* rwx protection bit */
02709 #define X_BIT               0000001 /* rwx protection bit */
02710 #define I_NOT_ALLOC         0000000 /* this inode is free */
02711
02712 /* Flag used only in flags argument of dev_open. */
02713 #define RO_BIT              0200000 /* Open device readonly; fail if writable. */
02714

```

```

02715 /* Some limits. */
02716 #define MAX_BLOCK_NR ((block_t) 077777777)      /* largest block number */
02717 #define HIGHEST_ZONE  ((zone_t) 077777777)      /* largest zone number */
02718 #define MAX_INODE_NR ((ino_t) 037777777777)      /* largest inode number */
02719 #define MAX_FILE_POS ((off_t) 037777777777)      /* largest legal file offset */
02720
02721 #define NO_BLOCK          ((block_t) 0)           /* absence of a block number */
02722 #define NO_ENTRY          ((ino_t) 0)           /* absence of a dir entry */
02723 #define NO_ZONE            ((zone_t) 0)           /* absence of a zone number */
02724 #define NO_DEV             ((dev_t) 0)           /* absence of a device numb */

+++++include/minix/type.h
+++++include/minix/sysconfig.h

02800 #ifndef _TYPE_H
02801 #define _TYPE_H
02802
02803 #ifndef _MINIX_SYS_CONFIG_H
02804 #include <minix/sys_config.h>
02805 #endif
02806
02807 #ifndef _TYPES_H
02808 #include <sys/types.h>
02809 #endif
02810
02811 /* Type definitions. */
02812 typedef unsigned int vir_clicks;           /* virtual addr/length in clicks */
02813 typedef unsigned long phys_bytes;         /* physical addr/length in bytes */
02814 typedef unsigned int phys_clicks;          /* physical addr/length in clicks */
02815
02816 #if (_MINIX_CHIP == _CHIP_INTEL)
02817 typedef unsigned int vir_bytes; /* virtual addresses and lengths in bytes */
02818 #endif
02819
02820 #if (_MINIX_CHIP == _CHIP_M68000)
02821 typedef unsigned long vir_bytes; /* virtual addresses and lengths in bytes */
02822 #endif
02823
02824 #if (_MINIX_CHIP == _CHIP_SPARC)
02825 typedef unsigned long vir_bytes; /* virtual addresses and lengths in bytes */
02826 #endif
02827
02828 /* Memory map for local text, stack, data segments. */
02829 struct mem_map {
02830     vir_clicks mem_vir;           /* virtual address */
02831     phys_clicks mem_phys;        /* physical address */
02832     vir_clicks mem_len;          /* length */
02833 };
02834
02835 /* Memory map for remote memory areas, e.g., for the RAM disk. */
02836 struct far_mem {
02837     int in_use;                 /* entry in use, unless zero */
02838     phys_clicks mem_phys;        /* physical address */
02839     vir_clicks mem_len;          /* length */
02840 };
02841
02842 /* Structure for virtual copying by means of a vector with requests. */
02843 struct vir_addr {
02844     int proc_nr;                /* */
02845     int segment;                /* */
02846     vir_bytes offset;           /* */
02847 };
02848
02849 #define phys_cp_req vir_cp_req
02850 struct vir_cp_req {
02851     struct vir_addr src;
02852     struct vir_addr dst;

```

```

02853     phys_bytes count;
02854 };
02855
02856     typedef struct {
02857         vir_bytes iov_addr;           /* address of an I/O buffer */
02858         vir_bytes iov_size;          /* sizeof an I/O buffer */
02859     } iovec_t;
02860
02861     /* PM passes the address of a structure of this type to KERNEL when
02862     * sys_sendsig() is invoked as part of the signal catching mechanism.
02863     * The structure contain all the information that KERNEL needs to build
02864     * the signal stack.
02865     */
02866     struct sigmsg {
02867         int sm_signo;                /* signal number being caught */
02868         unsigned long sm_mask;        /* mask to restore when handler returns */
02869         vir_bytes sm_sighandler;      /* address of handler */
02870         vir_bytes sm_sigreturn;       /* address of __sigreturn in C library */
02871         vir_bytes sm_stkptr;          /* user stack pointer */
02872     };
02873
02874     /* This is used to obtain system information through SYS_GETINFO. */
02875     struct kinfo {
02876         phys_bytes code_base;         /* base of kernel code */
02877         phys_bytes code_size;
02878         phys_bytes data_base;         /* base of kernel data */
02879         phys_bytes data_size;
02880         vir_bytes proc_addr;          /* virtual address of process table */
02881         phys_bytes kmem_base;          /* kernel memory layout (/dev/kmem) */
02882         phys_bytes kmem_size;
02883         phys_bytes bootdev_base;       /* boot device from boot image (/dev/boot) */
02884         phys_bytes bootdev_size;
02885         phys_bytes bootdev_mem;
02886         phys_bytes params_base;        /* parameters passed by boot monitor */
02887         phys_bytes params_size;
02888         int nr_procs;                /* number of user processes */
02889         int nr_tasks;                /* number of kernel tasks */
02890         char release[6];              /* kernel release number */
02891         char version[6];              /* kernel version number */
02892         int relocking;               /* relocking check (for debugging) */
02893     };
02894
02895     struct machine {
02896         int pc_at;
02897         int ps_mca;
02898         int processor;
02899         int protected;
02900         int vdu_ega;
02901         int vdu_vga;
02902     };
02903
02904 #endif /* _TYPE_H */
+
+=====
+===== include/minix/ ipc.h
+=====+
03000 #ifndef _IPC_H
03001 #define _IPC_H
03002
03003 /*=====
03004     * Types relating to messages.
03005     *=====
03006
03007 #define M1             1
03008 #define M3             3
03009 #define M4             4
03010 #define M3_STRING        14

```

```

03011
03012     typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
03013     typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
03014     typedef struct {int m3i1, m3i2; char *m3p1; char m3cal[M3_STRING];} mess_3;
03015     typedef struct {long m4i1, m4i2, m4i3, m4i4, m4i5;} mess_4;
03016     typedef struct {short m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;} mess_5;
03017     typedef struct {int m7i1, m7i2, m7i3, m7i4; char *m7p1, *m7p2;} mess_7;
03018     typedef struct {int m8i1, m8i2; char *m8p1, *m8p2, *m8p3, *m8p4;} mess_8;
03019
03020     typedef struct {
03021         int m_source;                      /* who sent the message */
03022         int m_type;                        /* what kind of message is it */
03023         union {
03024             mess_1 m_m1;
03025             mess_2 m_m2;
03026             mess_3 m_m3;
03027             mess_4 m_m4;
03028             mess_5 m_m5;
03029             mess_7 m_m7;
03030             mess_8 m_m8;
03031         } m_u;
03032     } message;
03033
03034     /* The following defines provide names for useful members. */
03035 #define m1_i1 m_u.m_m1.m1i1
03036 #define m1_i2 m_u.m_m1.m1i2
03037 #define m1_i3 m_u.m_m1.m1i3
03038 #define m1_p1 m_u.m_m1.m1p1
03039 #define m1_p2 m_u.m_m1.m1p2
03040 #define m1_p3 m_u.m_m1.m1p3
03041
03042 #define m2_i1 m_u.m_m2.m2i1
03043 #define m2_i2 m_u.m_m2.m2i2
03044 #define m2_i3 m_u.m_m2.m2i3
03045 #define m2_l1 m_u.m_m2.m2l1
03046 #define m2_l2 m_u.m_m2.m2l2
03047 #define m2_p1 m_u.m_m2.m2p1
03048
03049 #define m3_i1 m_u.m_m3.m3i1
03050 #define m3_i2 m_u.m_m3.m3i2
03051 #define m3_p1 m_u.m_m3.m3p1
03052 #define m3_cal m_u.m_m3.m3cal
03053
03054 #define m4_l1 m_u.m_m4.m4l1
03055 #define m4_l2 m_u.m_m4.m4l2
03056 #define m4_l3 m_u.m_m4.m4l3
03057 #define m4_l4 m_u.m_m4.m4l4
03058 #define m4_l5 m_u.m_m4.m4l5
03059
03060 #define m5_c1 m_u.m_m5.m5c1
03061 #define m5_c2 m_u.m_m5.m5c2
03062 #define m5_i1 m_u.m_m5.m5i1
03063 #define m5_i2 m_u.m_m5.m5i2
03064 #define m5_l1 m_u.m_m5.m5l1
03065 #define m5_l2 m_u.m_m5.m5l2
03066 #define m5_l3 m_u.m_m5.m5l3
03067
03068 #define m7_i1 m_u.m_m7.m7i1
03069 #define m7_i2 m_u.m_m7.m7i2
03070 #define m7_i3 m_u.m_m7.m7i3
03071 #define m7_i4 m_u.m_m7.m7i4
03072 #define m7_p1 m_u.m_m7.m7p1
03073 #define m7_p2 m_u.m_m7.m7p2
03074
03075 #define m8_i1 m_u.m_m8.m8i1
03076 #define m8_i2 m_u.m_m8.m8i2
03077 #define m8_p1 m_u.m_m8.m8p1

```

```

03078 #define m8_p2 m_u.m_m8.m8p2
03079 #define m8_p3 m_u.m_m8.m8p3
03080 #define m8_p4 m_u.m_m8.m8p4
03081
03082 /*=====
03083 * Minix run-time system (IPC).
03084 *=====
03085
03086 /* Hide names to avoid name space pollution. */
03087 #define echo _echo
03088 #define notify _notify
03089 #define sendrec _sendrec
03090 #define receive _receive
03091 #define send _send
03092 #define nb_receive _nb_receive
03093 #define nb_send _nb_send
03094
03095 _PROTOTYPE( int echo, (message *m_ptr) );
03096 _PROTOTYPE( int notify, (int dest) );
03097 _PROTOTYPE( int sendrec, (int src_dest, message *m_ptr) );
03098 _PROTOTYPE( int receive, (int src, message *m_ptr) );
03099 _PROTOTYPE( int send, (int dest, message *m_ptr) );
03100 _PROTOTYPE( int nb_receive, (int src, message *m_ptr) );
03101 _PROTOTYPE( int nb_send, (int dest, message *m_ptr) );
03102
03103 #endif /* _IPC_H */

+++++
#include/minix/syslib.h
+++++

03200 /* Prototypes for system library functions. */
03201
03202 #ifndef _SYSLIB_H
03203 #define _SYSLIB_H
03204
03205 #ifndef _TYPES_H
03206 #include <sys/types.h>
03207 #endif
03208
03209 #ifndef _IPC_H
03210 #include <minix/ ipc.h>
03211 #endif
03212
03213 #ifndef _DEVIO_H
03214 #include <minix/devio.h>
03215 #endif
03216
03217 /* Forward declaration */
03218 struct reg86u;
03219
03220 #define SYSTASK SYSTEM
03221
03222 /*=====
03223 * Minix system library.
03224 *=====
03225 _PROTOTYPE( int _taskcall, (int who, int syscallnr, message *msgptr));
03226
03227 _PROTOTYPE( int sys_abort, (int how, ...));
03228 _PROTOTYPE( int sys_exec, (int proc, char *ptr,
03229                           char *aout, vir_bytes initpc));
03230 _PROTOTYPE( int sys_fork, (int parent, int child));
03231 _PROTOTYPE( int sys_newmap, (int proc, struct mem_map *ptr));
03232 _PROTOTYPE( int sys_exit, (int proc));
03233 _PROTOTYPE( int sys_trace, (int req, int proc, long addr, long *data_p));
03234

```

```

03235 _PROTOTYPE( int sys_svrctl, (int proc, int req, int priv,vir_bytes argp));
03236 _PROTOTYPE( int sys_nice, (int proc, int priority));
03237
03238 _PROTOTYPE( int sys_int86, (struct reg86u *reg86p));
03239
03240 /* Shorthands for sys_sdevio() system call. */
03241 #define sys_insb(port, proc_nr, buffer, count) \
03242     sys_sdevio(DIO_INPUT, port, DIO_BYTE, proc_nr, buffer, count)
03243 #define sys_insw(port, proc_nr, buffer, count) \
03244     sys_sdevio(DIO_INPUT, port, DIO_WORD, proc_nr, buffer, count)
03245 #define sys_outsb(port, proc_nr, buffer, count) \
03246     sys_sdevio(DIO_OUTPUT, port, DIO_BYTE, proc_nr, buffer, count)
03247 #define sys_outsw(port, proc_nr, buffer, count) \
03248     sys_sdevio(DIO_OUTPUT, port, DIO_WORD, proc_nr, buffer, count)
03249 _PROTOTYPE( int sys_sdevio, (int req, long port, int type, int proc_nr,
03250             void *buffer, int count));
03251
03252 /* Clock functionality: get system times or (un)schedule an alarm call. */
03253 _PROTOTYPE( int sys_times, (int proc_nr, clock_t *ptr));
03254 _PROTOTYPE( int sys_setalarm, (clock_t exp_time, int abs_time));
03255
03256 /* Shorthands for sys_irqctl() system call. */
03257 #define sys_irqdisable(hook_id) \
03258     sys_irqctl(IRQ_DISABLE, 0, 0, hook_id)
03259 #define sys_irqenable(hook_id) \
03260     sys_irqctl(IRQ_ENABLE, 0, 0, hook_id)
03261 #define sys_irqsetpolicy(irq_vec, policy, hook_id) \
03262     sys_irqctl(IRQ_SETPOLICY, irq_vec, policy, hook_id)
03263 #define sys_irqrmpolicy(irq_vec, hook_id) \
03264     sys_irqctl(IRQ_RMPOLICY, irq_vec, 0, hook_id)
03265 _PROTOTYPE( int sys_irqctl, (int request, int irq_vec, int policy,
03266             int *irq_hook_id));
03267
03268 /* Shorthands for sys_vircopy() and sys_physcopy() system calls. */
03269 #define sys_biosin(bios_vir, dst_vir, bytes) \
03270     sys_vircopy(SELF, BIOS_SEG, bios_vir, SELF, D, dst_vir, bytes)
03271 #define sys_biosout(src_vir, bios_vir, bytes) \
03272     sys_vircopy(SELF, D, src_vir, SELF, BIOS_SEG, bios_vir, bytes)
03273 #define sys_datacopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
03274     sys_vircopy(src_proc, D, src_vir, dst_proc, D, dst_vir, bytes)
03275 #define sys_textcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
03276     sys_vircopy(src_proc, T, src_vir, dst_proc, T, dst_vir, bytes)
03277 #define sys_stackcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
03278     sys_vircopy(src_proc, S, src_vir, dst_proc, S, dst_vir, bytes)
03279 _PROTOTYPE( int sys_vircopy, (int src_proc, int src_seg, vir_bytes src_vir,
03280             int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));
03281
03282 #define sys_abscopy(src_phys, dst_phys, bytes) \
03283     sys_physcopy(NONE, PHYS_SEG, src_phys, NONE, PHYS_SEG, dst_phys, bytes)
03284 _PROTOTYPE( int sys_physcopy, (int src_proc, int src_seg, vir_bytes src_vir,
03285             int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));
03286 _PROTOTYPE( int sys_memset, (unsigned long pattern,
03287             phys_bytes base, phys_bytes bytes));
03288
03289 /* Vectored virtual / physical copy calls. */
03290 #if DEAD_CODE           /* library part not yet implemented */
03291 _PROTOTYPE( int sys_virvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
03292 _PROTOTYPE( int sys_physvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
03293 #endif
03294
03295 _PROTOTYPE( int sys_umap, (int proc_nr, int seg, vir_bytes vir_addr,
03296             vir_bytes bytes, phys_bytes *phys_addr));
03297 _PROTOTYPE( int sys_segctl, (int *index, u16_t *seg, vir_bytes *off,
03298             phys_bytes phys, vir_bytes size));
03299
03300 /* Shorthands for sys_getinfo() system call. */
03301 #define sys_getkmessages(dst)    sys_getinfo(GET_KMESSAGES, dst, 0,0,0)

```

```

03302 #define sys_getkinfo(dst)      sys_getinfo(GET_KINFO, dst, 0,0,0)
03303 #define sys_getmachine(dst)    sys_getinfo(GET_MACHINE, dst, 0,0,0)
03304 #define sys_getproctab(dst)    sys_getinfo(GET_PROCTAB, dst, 0,0,0)
03305 #define sys_getprivtab(dst)    sys_getinfo(GET_PRIVTAB, dst, 0,0,0)
03306 #define sys_getproc(dst,nr)    sys_getinfo(GET_PROC, dst, 0,0, nr)
03307 #define sys_getrandomness(dst)  sys_getinfo(GET_RANDOMNESS, dst, 0,0,0)
03308 #define sys_getimage(dst)       sys_getinfo(GET_IMAGE, dst, 0,0,0)
03309 #define sys_getirqhooks(dst)   sys_getinfo(GET_IROHOOKS, dst, 0,0,0)
03310 #define sys_getmonparams(v,v1)  sys_getinfo(GET_MONPARAMS, v,v1, 0,0)
03311 #define sys_getschedinfo(v1,v2) sys_getinfo(GET_SCHEDINFO, v1,0, v2,0)
03312 #define sys_getlocktimings(dst) sys_getinfo(GET_LOCKTIMING, dst, 0,0,0)
03313 #define sys_getbiosbuffer(virp, sizep) sys_getinfo(GET_BIOSBUFFER, virp, \
03314           sizeof(*virp), sizep, sizeof(*sizep))
03315 _PROTOTYPE(int sys_getinfo, (int request, void *val_ptr, int val_len,
03316                           void *val_ptr2, int val_len2)) ;
03317
03318 /* Signal control. */
03319 _PROTOTYPE(int sys_kill, (int proc, int sig) );
03320 _PROTOTYPE(int sys_sigsend, (int proc_nr, struct sigmsg *sig_ctxt) );
03321 _PROTOTYPE(int sys_sigreturn, (int proc_nr, struct sigmsg *sig_ctxt) );
03322 _PROTOTYPE(int sys_getksig, (int *k_proc_nr, sigset_t *k_sig_map) );
03323 _PROTOTYPE(int sys_endksig, (int proc_nr) );
03324
03325 /* NOTE: two different approaches were used to distinguish the device I/O
03326 * types 'byte', 'word': the latter uses #define and results in a
03327 * smaller implementation, but looses the static type checking.
03328 */
03329 _PROTOTYPE(int sys_voutb, (pvb_pair_t *pvb_pairs, int nr_ports) );
03330 _PROTOTYPE(int sys_voutw, (pwv_pair_t *pwv_pairs, int nr_ports) );
03331 _PROTOTYPE(int sys_voutl, (pvl_pair_t *pvl_pairs, int nr_ports) );
03332 _PROTOTYPE(int sys_vinb, (pvb_pair_t *pvb_pairs, int nr_ports) );
03333 _PROTOTYPE(int sys_vinw, (pwv_pair_t *pwv_pairs, int nr_ports) );
03334 _PROTOTYPE(int sys_vinl, (pvl_pair_t *pvl_pairs, int nr_ports) );
03335
03336 /* Shorthands for sys_out() system call. */
03337 #define sys_outb(p,v)  sys_out((p), (unsigned long) (v), DIO_BYTE)
03338 #define sys_outw(p,v)  sys_out((p), (unsigned long) (v), DIO_WORD)
03339 #define sys_outl(p,v)  sys_out((p), (unsigned long) (v), DIO_LONG)
03340 _PROTOTYPE(int sys_out, (int port, unsigned long value, int type) );
03341
03342 /* Shorthands for sys_in() system call. */
03343 #define sys_inb(p,v)   sys_in((p), (unsigned long*) (v), DIO_BYTE)
03344 #define sys_inw(p,v)   sys_in((p), (unsigned long*) (v), DIO_WORD)
03345 #define sys_inl(p,v)   sys_in((p), (unsigned long*) (v), DIO_LONG)
03346 _PROTOTYPE(int sys_in, (int port, unsigned long *value, int type) );
03347
03348 #endif /* _SYSLIB_H */
03349

+++++
include/minix/sysutil.h
+++++
03400 #ifndef _EXTRALIB_H
03401 #define _EXTRALIB_H
03402
03403 /* Extra system library definitions to support device drivers and servers.
03404 *
03405 * Created:
03406 *     Mar 15, 2004 by Jorrit N. Herder
03407 *
03408 * Changes:
03409 *     May 31, 2005: added printf, kputc (relocated from syslib)
03410 *     May 31, 2005: added gettimeofday
03411 *     Mar 18, 2005: added tickdelay
03412 *     Oct 01, 2004: added env_parse, env_prefix, env_panic
03413 *     Jul 13, 2004: added fkey_ctl
03414 *     Apr 28, 2004: added report, panic

```

```

03415 *      Mar 31, 2004: setup like other libraries, such as syslib
03416 */
03417
03418 /*=====
03419 * Miscellaneous helper functions.
03420 *=====
03421
03422 /* Environment parsing return values. */
03423 #define EP_BUF_SIZE 128      /* local buffer for env value */
03424 #define EP_UNSET 0          /* variable not set */
03425 #define EP_OFF 1           /* var = off */
03426 #define EP_ON 2            /* var = on (or field left blank) */
03427 #define EP_SET 3           /* var = 1:2:3 (nonblank field) */
03428 #define EP_EGETENV 4        /* sys_getenv() failed ... */
03429
03430 _PROTOTYPE( void env_setargs, (int argc, char *argv[]) );
03431 _PROTOTYPE( int env_get_param, (char *key, char *value, int max_size) );
03432 _PROTOTYPE( int env_prefix, (char *env, char *prefix) );
03433 _PROTOTYPE( void env_panic, (char *key) );
03434 _PROTOTYPE( int env_parse, (char *env, char *fmt, int field, long *param,
03435           long min, long max) );
03436
03437 #define fkey_map(fkeys, sfkeys) fkey_ctl(FKEY_MAP, (fkeys), (sfkeys))
03438 #define fkey_unmap(fkeys, sfkeys) fkey_ctl(FKEY_UNMAP, (fkeys), (sfkeys))
03439 #define fkey_events(fkeys, sfkeys) fkey_ctl(FKEY_EVENTS, (fkeys), (sfkeys))
03440 _PROTOTYPE( int fkey_ctl, (int req, int *fkeys, int *sfkeys) );
03441
03442 _PROTOTYPE( int printf, (const char *fmt, ...) );
03443 _PROTOTYPE( void kputc, (int c));
03444 _PROTOTYPE( void report, (char *who, char *mess, int num));
03445 _PROTOTYPE( void panic, (char *who, char *mess, int num));
03446 _PROTOTYPE( int getuptime, (clock_t *ticks));
03447 _PROTOTYPE( int tickdelay, (clock_t ticks));
03448
03449 #endif /* _EXTRALIB_H */
03450

```

```

+++++
include/minix/callnr.h
+++++

```

```

03500 #define NCALLS         91    /* number of system calls allowed */
03501
03502 #define EXIT          1
03503 #define FORK          2
03504 #define READ          3
03505 #define WRITE         4
03506 #define OPEN          5
03507 #define CLOSE         6
03508 #define WAIT          7
03509 #define CREAT         8
03510 #define LINK          9
03511 #define UNLINK        10
03512 #define WAITPID       11
03513 #define CHDIR         12
03514 #define TIME          13
03515 #define MKNOD         14
03516 #define CHMOD         15
03517 #define CHOWN         16
03518 #define BRK           17
03519 #define STAT          18
03520 #define LSEEK          19
03521 #define GETPID        20
03522 #define MOUNT         21

```

```

03523 #define UMOUNT 22
03524 #define SETUID 23
03525 #define GETUID 24
03526 #define STIME 25
03527 #define PTRACE 26
03528 #define ALARM 27
03529 #define FSTAT 28
03530 #define PAUSE 29
03531 #define UTIME 30
03532 #define ACCESS 33
03533 #define SYNC 36
03534 #define KILL 37
03535 #define RENAME 38
03536 #define MKDIR 39
03537 #define RMDIR 40
03538 #define DUP 41
03539 #define PIPE 42
03540 #define TIMES 43
03541 #define SETGID 46
03542 #define GETGID 47
03543 #define SIGNAL 48
03544 #define IOCTL 54
03545 #define FNTL 55
03546 #define EXEC 59
03547 #define UMASK 60
03548 #define CHROOT 61
03549 #define SETSID 62
03550 #define GETPGRP 63
03551
03552 /* The following are not system calls, but are processed like them. */
03553 #define UNPAUSE 65 /* to MM or FS: check for EINTR */
03554 #define REVIVE 67 /* to FS: revive a sleeping process */
03555 #define TASK_REPLY 68 /* to FS: reply code from tty task */
03556
03557 /* Posix signal handling. */
03558 #define SIGACTION 71
03559 #define SIGSUSPEND 72
03560 #define SIGPENDING 73
03561 #define SIGPROCMASK 74
03562 #define SIGRETURN 75
03563
03564 #define REBOOT 76 /* to PM */
03565
03566 /* MINIX specific calls, e.g., to support system services. */
03567 #define SVRCTL 77
03568
03569 #define GETSYSINFO 79 /* unused */
03570 #define GETPROCNR 80 /* to PM or FS */
03571 #define DEVCTL 81 /* to PM */
03572 #define FSTATFS 82 /* to FS */
03573 #define ALLOCMEM 83 /* to PM */
03574 #define FREEMEM 84 /* to PM */
03575 #define SELECT 85 /* to FS */
03576 #define FCHEDIR 86 /* to FS */
03577 #define FSYNC 87 /* to FS */
03578 #define GETPRIORITY 88 /* to PM */
03579 #define SETPRIORITY 89 /* to PM */
03580 #define GETTIMEOFDAY 90 /* to PM */

```

```
#####
include/minix/com.h
#####
```

```
03600 #ifndef _MINIX_COM_H
```

```

03601 #define _MINIX_COM_H
03602
03603 /*=====
03604 *          Magic process numbers
03605 *=====
03606
03607 #define ANY          0x7ace /* used to indicate 'any process' */
03608 #define NONE         0x6ace /* used to indicate 'no process at all' */
03609 #define SELF         0x8ace /* used to indicate 'own process' */
03610
03611 /*=====
03612 *          Process numbers of processes in the system image
03613 *=====
03614
03615 /* The values of several task numbers depend on whether they or other tasks
03616 * are enabled. They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
03617 * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
03618 * the same number as the previous task and is further unused. Note that the
03619 * order should correspond to the order in the task table defined in table.c.
03620 */
03621
03622 /* Kernel tasks. These all run in the same address space. */
03623 #define IDLE          -4    /* runs when no one else can run */
03624 #define CLOCK         -3    /* alarms and other clock functions */
03625 #define SYSTEM        -2    /* request system functionality */
03626 #define KERNEL        -1    /* pseudo-process for IPC and scheduling */
03627 #define HARDWARE      KERNEL /* for hardware interrupt handlers */
03628
03629 /* Number of tasks. Note that NR_PROCS is defined in <minix/config.h>. */
03630 #define NR_TASKS       4
03631
03632 /* User-space processes, that is, device drivers, servers, and INIT. */
03633 #define PM_PROC_NR     0    /* process manager */
03634 #define FS_PROC_NR     1    /* file system */
03635 #define RS_PROC_NR     2    /* reincarnation server */
03636 #define MEM_PROC_NR    3    /* memory driver (RAM disk, null, etc.) */
03637 #define LOG_PROC_NR    4    /* log device driver */
03638 #define TTY_PROC_NR    5    /* terminal (TTY) driver */
03639 #define DRVNR_PROC_NR  6    /* device driver for boot medium */
03640 #define INIT_PROC_NR   7    /* init -- goes multiuser */
03641
03642 /* Number of processes contained in the system image. */
03643 #define NR_BOOT_PROCS (NR_TASKS + INIT_PROC_NR + 1)
03644
03645 /*=====
03646 *          Kernel notification types
03647 *=====
03648
03649 /* Kernel notification types. In principle, these can be sent to any process,
03650 * so make sure that these types do not interfere with other message types.
03651 * Notifications are prioritized because of the way they are unhold() and
03652 * blocking notifications are delivered. The lowest numbers go first. The
03653 * offset are used for the per-process notification bit maps.
03654 */
03655 #define NOTIFY_MESSAGE      0x1000
03656 #define NOTIFY_FROM(p_nr)   (NOTIFY_MESSAGE | ((p_nr) + NR_TASKS))
03657 # define SYN_ALARM        NOTIFY_FROM(CLOCK) /* synchronous alarm */
03658 # define SYS_SIG          NOTIFY_FROM(SYSTEM) /* system signal */
03659 # define HARD_INT         NOTIFY_FROM(HARDWARE) /* hardware interrupt */
03660 # define NEW_KSIG          NOTIFY_FROM(HARDWARE) /* new kernel signal */
03661 # define FKEY_PRESSED     NOTIFY_FROM(TTY_PROC_NR)/* function key press */
03662
03663 /* Shorthands for message parameters passed with notifications. */
03664 #define NOTIFY_SOURCE      m_source
03665 #define NOTIFY_TYPE        m_type
03666 #define NOTIFY_ARG         m2_11
03667 #define NOTIFY_TIMESTAMP   m2_12

```

```

03668 #define NOTIFY_FLAGS          m2_i1
03669
03670 /*=====
03671 *      Messages for BLOCK and CHARACTER device drivers
03672 *=====
03673
03674 /* Message types for device drivers. */
03675 #define DEV_RQ_BASE   0x400    /* base for device request types */
03676 #define DEV_RS_BASE   0x500    /* base for device response types */
03677
03678 #define CANCEL        (DEV_RQ_BASE + 0) /* general req to force a task to cancel */
03679 #define DEV_READ       (DEV_RQ_BASE + 3) /* read from minor device */
03680 #define DEV_WRITE      (DEV_RQ_BASE + 4) /* write to minor device */
03681 #define DEV_IOCTL      (DEV_RQ_BASE + 5) /* I/O control code */
03682 #define DEV_OPEN        (DEV_RQ_BASE + 6) /* open a minor device */
03683 #define DEV_CLOSE       (DEV_RQ_BASE + 7) /* close a minor device */
03684 #define DEV_SCATTER     (DEV_RQ_BASE + 8) /* write from a vector */
03685 #define DEV_GATHER      (DEV_RQ_BASE + 9) /* read into a vector */
03686 #define TTY_SETPGRP     (DEV_RQ_BASE + 10) /* set process group */
03687 #define TTY_EXIT        (DEV_RQ_BASE + 11) /* process group leader exited */
03688 #define DEV_SELECT      (DEV_RQ_BASE + 12) /* request select() attention */
03689 #define DEV_STATUS      (DEV_RQ_BASE + 13) /* request driver status */
03690
03691 #define DEV_REPLY       (DEV_RS_BASE + 0) /* general task reply */
03692 #define DEV_CLONED      (DEV_RS_BASE + 1) /* return cloned minor */
03693 #define DEV_REVIVE      (DEV_RS_BASE + 2) /* driver revives process */
03694 #define DEV_IO_READY    (DEV_RS_BASE + 3) /* selected device ready */
03695 #define DEV_NO_STATUS   (DEV_RS_BASE + 4) /* empty status reply */
03696
03697 /* Field names for messages to block and character device drivers. */
03698 #define DEVICE          m2_i1 /* major-minor device */
03699 #define PROC_NR         m2_i2 /* which (proc) wants I/O? */
03700 #define COUNT            m2_i3 /* how many bytes to transfer */
03701 #define REQUEST          m2_i3 /* ioctl request code */
03702 #define POSITION         m2_l1 /* file offset */
03703 #define ADDRESS          m2_p1 /* core buffer address */
03704
03705 /* Field names for DEV_SELECT messages to device drivers. */
03706 #define DEV_MINOR        m2_i1 /* minor device */
03707 #define DEV_SEL_OPS      m2_i2 /* which select operations are requested */
03708 #define DEV_SEL_WATCH    m2_i3 /* request notify if no operations are ready */
03709
03710 /* Field names used in reply messages from tasks. */
03711 #define REP_PROC_NR      m2_i1 /* # of proc on whose behalf I/O was done */
03712 #define REP_STATUS       m2_i2 /* bytes transferred or error number */
03713 # define SUSPEND        -998 /* status to suspend caller, reply later */
03714
03715 /* Field names for messages to TTY driver. */
03716 #define TTY_LINE          DEVICE /* message parameter: terminal line */
03717 #define TTY_REQUEST       COUNT /* message parameter: ioctl request code */
03718 #define TTY_SPEK          POSITION/* message parameter: ioctl speed, erasing */
03719 #define TTY_FLAGS          m2_l2 /* message parameter: ioctl tty mode */
03720 #define TTY_PGRP          m2_i3 /* message parameter: process group */
03721
03722 /* Field names for the QIC 02 status reply from tape driver */
03723 #define TAPE_STAT0        m2_l1
03724 #define TAPE_STAT1        m2_l2
03725
03726 /*=====
03727 *      Messages for networking layer
03728 *=====
03729
03730 /* Message types for network layer requests. This layer acts like a driver. */
03731 #define NW_OPEN           DEV_OPEN
03732 #define NW_CLOSE          DEV_CLOSE
03733 #define NW_READ           DEV_READ
03734 #define NW_WRITE          DEV_WRITE

```

```

03735 #define NW_IOCTL      DEV_IOCTL
03736 #define NW_CANCEL     CANCEL
03737
03738 /* Base type for data link layer requests and responses. */
03739 #define DL_RQ_BASE    0x800
03740 #define DL_RS_BASE    0x900
03741
03742 /* Message types for data link layer requests. */
03743 #define DL_WRITE       (DL_RQ_BASE + 3)
03744 #define DL_WRITEV      (DL_RQ_BASE + 4)
03745 #define DL_READ        (DL_RQ_BASE + 5)
03746 #define DL_READV       (DL_RQ_BASE + 6)
03747 #define DL_INIT        (DL_RQ_BASE + 7)
03748 #define DL_STOP        (DL_RQ_BASE + 8)
03749 #define DL_GETSTAT     (DL_RQ_BASE + 9)
03750
03751 /* Message type for data link layer replies. */
03752 #define DL_INIT_REPLY   (DL_RS_BASE + 20)
03753 #define DL_TASK_REPLY   (DL_RS_BASE + 21)
03754
03755 /* Field names for data link layer messages. */
03756 #define DL_PORT         m2_i1
03757 #define DL_PROC         m2_i2
03758 #define DL_COUNT        m2_i3
03759 #define DL_MODE         m2_i11
03760 #define DL_CLK          m2_i12
03761 #define DL_ADDR         m2_p1
03762 #define DL_STAT         m2_i11
03763
03764 /* Bits in 'DL_STAT' field of DL replies. */
03765 # define DL_PACK_SEND 0x01
03766 # define DL_PACK_RECV 0x02
03767 # define DL_READ_IP   0x04
03768
03769 /* Bits in 'DL_MODE' field of DL requests. */
03770 # define DL_NOMODE    0x0
03771 # define DL_PROMISC_REQ 0x2
03772 # define DL_MULTI_REQ  0x4
03773 # define DL_BROAD_REQ  0x8
03774
03775 /*=====
03776 *           SYSTASK request types and field names
03777 *=====
03778
03779 /* System library calls are dispatched via a call vector, so be careful when
0380  * modifying the system call numbers. The numbers here determine which call
0381  * is made from the call vector.
0382  */
0383 #define KERNEL_CALL    0x600 /* base for kernel calls to SYSTEM */
0384
0385 # define SYS_FORK      (KERNEL_CALL + 0)   /* sys_fork() */
0386 # define SYS_EXEC      (KERNEL_CALL + 1)   /* sys_exec() */
0387 # define SYS_EXIT      (KERNEL_CALL + 2)   /* sys_exit() */
0388 # define SYS_NICE      (KERNEL_CALL + 3)   /* sys_nice() */
0389 # define SYS_PRIVCTL   (KERNEL_CALL + 4)   /* sys_privctl() */
0390 # define SYS_TRACE     (KERNEL_CALL + 5)   /* sys_trace() */
0391 # define SYS_KILL      (KERNEL_CALL + 6)   /* sys_kill() */
0392
0393 # define SYS_GETKSIG   (KERNEL_CALL + 7)   /* sys_getsig() */
0394 # define SYS_ENDKSIG   (KERNEL_CALL + 8)   /* sys_endsig() */
0395 # define SYS_SIGSEND   (KERNEL_CALL + 9)   /* sys_sigsend() */
0396 # define SYS_SIGRETURN (KERNEL_CALL + 10)  /* sys_sigreturn() */
0397
0398 # define SYS_NEWMAP    (KERNEL_CALL + 11)  /* sys_newmap() */
0399 # define SYS_SEGCTL   (KERNEL_CALL + 12)  /* sys_segctl() */
0400 # define SYS_MEMSET   (KERNEL_CALL + 13)  /* sys_memset() */
0401

```

```

03802 # define SYS_UMAP (KERNEL_CALL + 14) /* sys_umap() */
03803 # define SYS_VIRCOPY (KERNEL_CALL + 15) /* sys_vircopy() */
03804 # define SYS_PHYSCOPY (KERNEL_CALL + 16) /* sys_physcopy() */
03805 # define SYS_VIRVCOPY (KERNEL_CALL + 17) /* sys_virvcopy() */
03806 # define SYS_PHYSVCOPY (KERNEL_CALL + 18) /* sys_physvcopy() */
03807
03808 # define SYS_IRQCTL (KERNEL_CALL + 19) /* sys_irqctl() */
03809 # define SYS_INT86 (KERNEL_CALL + 20) /* sys_int86() */
03810 # define SYS_DEVIO (KERNEL_CALL + 21) /* sys_devio() */
03811 # define SYS_SDEVIO (KERNEL_CALL + 22) /* sys_sdevio() */
03812 # define SYS_VDEVIO (KERNEL_CALL + 23) /* sys_vdevio() */
03813
03814 # define SYS_SETALARM (KERNEL_CALL + 24) /* sys_setalarm() */
03815 # define SYS_TIMES (KERNEL_CALL + 25) /* sys_times() */
03816 # define SYS_GETINFO (KERNEL_CALL + 26) /* sys_getinfo() */
03817 # define SYS_ABORT (KERNEL_CALL + 27) /* sys_abort() */
03818
03819 #define NR_SYS_CALLS 28 /* number of system calls */
03820
03821 /* Field names for SYS_MEMSET, SYS_SEGCTL. */
03822 #define MEM_PTR m2_p1 /* base */
03823 #define MEM_COUNT m2_l1 /* count */
03824 #define MEM_PATTERN m2_l2 /* pattern to write */
03825 #define MEM_CHUNK_BASE m4_l1 /* physical base address */
03826 #define MEM_CHUNK_SIZE m4_l2 /* size of mem chunk */
03827 #define MEM_TOT_SIZE m4_l3 /* total memory size */
03828 #define MEM_CHUNK_TAG m4_l4 /* tag to identify chunk of mem */
03829
03830 /* Field names for SYS_DEVIO, SYS_VDEVIO, SYS_SDEVIO. */
03831 #define DIO_REQUEST m2_i3 /* device in or output */
03832 # define DIO_INPUT 0 /* input */
03833 # define DIO_OUTPUT 1 /* output */
03834 #define DIO_TYPE m2_i1 /* flag indicating byte, word, or long */
03835 # define DIO_BYTE 'b' /* byte type values */
03836 # define DIO_WORD 'w' /* word type values */
03837 # define DIO_LONG 'l' /* long type values */
03838 #define DIO_PORT m2_l1 /* single port address */
03839 #define DIO_VALUE m2_l2 /* single I/O value */
03840 #define DIO_VEC_ADDR m2_p1 /* address of buffer or (p,v)-pairs */
03841 #define DIO_VEC_SIZE m2_l2 /* number of elements in vector */
03842 #define DIO_VEC_PROC m2_i2 /* number of process where vector is */
03843
03844 /* Field names for SYS_SIGNALRM, SYS_FLAGARLM, SYS_SYNCALRM. */
03845 #define ALRM_EXP_TIME m2_l1 /* expire time for the alarm call */
03846 #define ALRM_ABS_TIME m2_i2 /* set to 1 to use absolute alarm time */
03847 #define ALRM_TIME_LEFT m2_l1 /* how many ticks were remaining */
03848 #define ALRM_PROC_NR m2_i1 /* which process wants the alarm? */
03849 #define ALRM_FLAG_PTR m2_p1 /* virtual address of timeout flag */
03850
03851 /* Field names for SYS_IRQCTL. */
03852 #define IRQ_REQUEST m5_c1 /* what to do? */
03853 # define IRQ_SETPOLICY 1 /* manage a slot of the IRQ table */
03854 # define IRQ_RMPOLICY 2 /* remove a slot of the IRQ table */
03855 # define IRQ_ENABLE 3 /* enable interrupts */
03856 # define IRQ_DISABLE 4 /* disable interrupts */
03857 #define IRQ_VECTOR m5_c2 /* irq vector */
03858 #define IRQ_POLICY m5_i1 /* options for IRQCTL request */
03859 # define IRQ_REENABLE 0x001 /* reenable IRQ line after interrupt */
03860 # define IRQ_BYTE 0x100 /* byte values */
03861 # define IRQ_WORD 0x200 /* word values */
03862 # define IRQ_LONG 0x400 /* long values */
03863 #define IRQ_PROC_NR m5_i2 /* process number, SELF, NONE */
03864 #define IRQ_HOOK_ID m5_l3 /* id of irq hook at kernel */
03865
03866 /* Field names for SYS_SEGCTL. */
03867 #define SEG_SELECT m4_l1 /* segment selector returned */
03868 #define SEG_OFFSET m4_l2 /* offset in segment returned */

```

```

03869 #define SEG_PHYS      m4_13 /* physical address of segment */
03870 #define SEG_SIZE       m4_14 /* segment size */
03871 #define SEG_INDEX      m4_15 /* segment index in remote map */
03872
03873 /* Field names for SYS_VIDCOPY. */
03874 #define VID_REQUEST     m4_11 /* what to do? */
03875 #define VID_VID_COPY    1 /* request vid_vid_copy() */
03876 #define MEM_VID_COPY    2 /* request mem_vid_copy() */
03877 #define VID_SRC_ADDR    m4_12 /* virtual address in memory */
03878 #define VID_SRC_OFFSET   m4_13 /* offset in video memory */
03879 #define VID_DST_OFFSET   m4_14 /* offset in video memory */
03880 #define VID_CP_COUNT    m4_15 /* number of words to be copied */
03881
03882 /* Field names for SYS_ABORT. */
03883 #define ABRT_HOW        m1_i1 /* RBT_REBOOT, RBT_HALT, etc. */
03884 #define ABRT_MON_PROC   m1_i2 /* process where monitor params are */
03885 #define ABRT_MON_LEN    m1_i3 /* length of monitor params */
03886 #define ABRT_MON_ADDR   m1_p1 /* virtual address of monitor params */
03887
03888 /* Field names for _UMAP, _VIRCOPY, _PHYSCOPY. */
03889 #define CP_SRC_SPACE    m5_c1 /* T or D space (stack is also D) */
03890 #define CP_SRC_PROC_NR  m5_i1 /* process to copy from */
03891 #define CP_SRC_ADDR     m5_11 /* address where data come from */
03892 #define CP_DST_SPACE    m5_c2 /* T or D space (stack is also D) */
03893 #define CP_DST_PROC_NR  m5_i2 /* process to copy to */
03894 #define CP_DST_ADDR     m5_12 /* address where data go to */
03895 #define CP_NR_BYTES     m5_13 /* number of bytes to copy */
03896
03897 /* Field names for SYS_VCOPY and SYS_VVIRCOPY. */
03898 #define VCP_NR_OK        m1_i2 /* number of successfull copies */
03899 #define VCP_VEC_SIZE     m1_i3 /* size of copy vector */
03900 #define VCP_VEC_ADDR     m1_p1 /* pointer to copy vector */
03901
03902 /* Field names for SYS_GETINFO. */
03903 #define I_REQUEST        m7_i3 /* what info to get */
03904 #define GET_KINFO         0 /* get kernel information structure */
03905 #define GET_IMAGE         1 /* get system image table */
03906 #define GET_PROCTAB       2 /* get kernel process table */
03907 #define GET_RANDOMNESS    3 /* get randomness buffer */
03908 #define GET_MONPARAMS    4 /* get monitor parameters */
03909 #define GET_KENV          5 /* get kernel environment string */
03910 #define GET_IRQHOOKS     6 /* get the IRQ table */
03911 #define GET_KMESSAGES    7 /* get kernel messages */
03912 #define GET_PRIVTAB       8 /* get kernel privileges table */
03913 #define GET_KADDRESSES   9 /* get various kernel addresses */
03914 #define GET_SCHEDINFO    10 /* get scheduling queues */
03915 #define GET_PROC          11 /* get process slot if given process */
03916 #define GET_MACHINE       12 /* get machine information */
03917 #define GET_LOCKTIMING   13 /* get lock()/unlock() latency timing */
03918 #define GET_BIOSBUFFER   14 /* get a buffer for BIOS calls */
03919 #define I_PROC_NR         m7_i4 /* calling process */
03920 #define I_VAL_PTR         m7_p1 /* virtual address at caller */
03921 #define I_VAL_LEN         m7_i1 /* max length of value */
03922 #define I_VAL_PTR2        m7_p2 /* second virtual address */
03923 #define I_VAL_LEN2        m7_i2 /* second length, or proc nr */
03924
03925 /* Field names for SYS_TIMES. */
03926 #define T_PROC_NR         m4_11 /* process to request time info for */
03927 #define T_USER_TIME       m4_11 /* user time consumed by process */
03928 #define T_SYSTEM_TIME     m4_12 /* system time consumed by process */
03929 #define T_CHILD_UTIME    m4_13 /* user time consumed by process' children */
03930 #define T_CHILD_STIME    m4_14 /* sys time consumed by process' children */
03931 #define T_BOOT_TICKS     m4_15 /* number of clock ticks since boot time */
03932
03933 /* Field names for SYS_TRACE, SYS_SVRCTL. */
03934 #define CTL_PROC_NR       m2_i1 /* process number of the caller */
03935 #define CTL_REQUEST        m2_i2 /* server control request */

```

```

03936 #define CTL_MM_PRIV      m2_i3    /* privilege as seen by PM */
03937 #define CTL_ARG_PTR      m2_p1    /* pointer to argument */
03938 #define CTL_ADDRESS       m2_l1    /* address at traced process' space */
03939 #define CTL_DATA          m2_l2    /* data field for tracing */
03940
03941 /* Field names for SYS_KILL, SYS_SIGCTL */
03942 #define SIG_REQUEST       m2_l2    /* PM signal control request */
03943 #define S_GETSIG          0        /* get pending kernel signal */
03944 #define S_ENDSIG          1        /* finish a kernel signal */
03945 #define S_SENDSIG         2        /* POSIX style signal handling */
03946 #define S_SIGRETURN        3        /* return from POSIX handling */
03947 #define S_KILL             4        /* servers kills process with signal */
03948 #define SIG_PROC           m2_i1    /* process number for inform */
03949 #define SIG_NUMBER         m2_i2    /* signal number to send */
03950 #define SIG_FLAGS          m2_i3    /* signal flags field */
03951 #define SIG_MAP            m2_l1    /* used by kernel to pass signal bit map */
03952 #define SIG_CTXT_PTR       m2_p1    /* pointer to info to restore signal context */
03953
03954 /* Field names for SYS_FORK, _EXEC, _EXIT, _NEWMAP. */
03955 #define PR_PROC_NR         m1_i1    /* indicates a (child) process */
03956 #define PR_PRIORITY         m1_i2    /* process priority */
03957 #define PR_PPROC_NR        m1_i2    /* indicates a (parent) process */
03958 #define PR_PID              m1_i3    /* process id at process manager */
03959 #define PR_STACK_PTR       m1_p1    /* used for stack ptr in sys_exec, sys_getsp */
03960 #define PR_TRACING          m1_i3    /* flag to indicate tracing is on/ off */
03961 #define PR_NAME_PTR         m1_p2    /* tells where program name is for dmp */
03962 #define PR_IP_PTR           m1_p3    /* initial value for ip after exec */
03963 #define PR_MEM_PTR          m1_p1    /* tells where memory map is for sys_newmap */
03964
03965 /* Field names for SYS_INT86 */
03966 #define INT86_REG86         m1_p1    /* pointer to registers */
03967
03968 /* Field names for SELECT (FS). */
03969 #define SEL_NFDS            m8_i1
03970 #define SEL_READFDS         m8_p1
03971 #define SEL_WRITEFDS        m8_p2
03972 #define SEL_ERRORFDS        m8_p3
03973 #define SEL_TIMEOUT         m8_p4
03974
03975 /***** * Messages for system management server * *****/
03976
03977 *=====
03978
03979 #define SRV_RQ_BASE          0x700
03980
03981 #define SRV_UP                (SRV_RQ_BASE + 0)      /* start system service */
03982 #define SRV_DOWN               (SRV_RQ_BASE + 1)      /* stop system service */
03983 #define SRV_STATUS              (SRV_RQ_BASE + 2)      /* get service status */
03984
03985 # define SRV_PATH_ADDR       m1_p1    /* path of binary */
03986 # define SRV_PATH_LEN        m1_i1    /* length of binary */
03987 # define SRV_ARGS_ADDR       m1_p2    /* arguments to be passed */
03988 # define SRV_ARGS_LEN        m1_i2    /* length of arguments */
03989 # define SRV_DEV_MAJOR        m1_i3    /* major device number */
03990 # define SRV_PRIV_ADDR       m1_p3    /* privileges string */
03991 # define SRV_PRIV_LEN        m1_i3    /* length of privileges */
03992
03993 /***** * Miscellaneous messages used by TTY * *****/
03994
03995 *=====
03996
03997 /* Miscellaneous request types and field names, e.g. used by IS server. */
03998 #define PANIC_DUMPS          97      /* debug dumps at the TTY on RBT_PANIC */
03999 #define FKEY_CONTROL          98      /* control a function key at the TTY */
04000 # define FKEY_REQUEST        m2_i1    /* request to perform at TTY */
04001 # define FKEY_MAP            10      /* observe function key */
04002 # define FKEY_UNMAP          11      /* stop observing function key */

```

```

04003 # define FKEY_EVENTS      12      /* request open key presses */
04004 # define FKEY_FKEYS        m2_l1    /* F1-F12 keys pressed */
04005 # define FKEY_SFKEYS      m2_l2    /* Shift-F1-F12 keys pressed */
04006 #define DIAGNOSTICS     100      /* output a string without FS in between */
04007 # define DIAG_PRINT_BUF    m1_p1
04008 # define DIAG_BUF_COUNT    m1_i1
04009 # define DIAG_PROC_NR      m1_i2
0410
0411 #endif /* _MINIX_COM_H */

+++++
include/minix/devio.h
+++++

04100 /* This file provides basic types and some constants for the
04101 * SYS_DEVIO and SYS_VDEVIO system calls, which allow user-level
04102 * processes to perform device I/O.
04103 *
04104 * Created:
04105 *     Apr 08, 2004 by Jorrit N. Herder
04106 */
04107
04108 #ifndef _DEVIO_H
04109 #define _DEVIO_H
04110
04111 #include <minix/sys_config.h>      /* needed to include <minix/type.h> */
04112 #include <sys/types.h>           /* u8_t, u16_t, u32_t needed */
04113
04114 typedef u16_t port_t;
04115 typedef U16_T Port_t;
04116
04117 /* We have different granularities of port I/O: 8, 16, 32 bits.
04118 * Also see <ibm/portio.h>, which has functions for bytes, words,
04119 * and longs. Hence, we need different (port,value)-pair types.
04120 */
04121 typedef struct { u16_t port; u8_t value; } pvb_pair_t;
04122 typedef struct { u16_t port; u16_t value; } pvw_pair_t;
04123 typedef struct { u16_t port; u32_t value; } pvl_pair_t;
04124
04125 /* Macro shorthand to set (port,value)-pair. */
04126 #define pv_set(pv, p, v) ((pv).port = (p), (pv).value = (v))
04127 #define pv_ptr_set(pv_ptr, p, v) ((pv_ptr)->port = (p), (pv_ptr)->value = (v))
04128
04129 #endif /* _DEVIO_H */

+++++
include/minix/dmap.h
+++++

04200 #ifndef _DMAP_H
04201 #define _DMAP_H
04202
04203 #include <minix/sys_config.h>
04204 #include <minix/ ipc.h>
04205
04206 /*=====
04207 *          Device <-> Driver Table
04208 *=====
04209
04210 /* Device table. This table is indexed by major device number. It provides
04211 * the link between major device numbers and the routines that process them.
04212 * The table can be update dynamically. The field 'dmap_flags' describe an
04213 * entry's current status and determines what control options are possible.
04214 */

```

```

04215 #define DMAP_MUTABLE          0x01    /* mapping can be overtaken */
04216 #define DMAP_BUSY           0x02    /* driver busy with request */
04217
04218 enum dev_style { STYLE_DEV, STYLE_NDEV, STYLE_TTY, STYLE_CLONE };
04219
04220 extern struct dmap {
04221     int _PROTOTYPE ((*dmap_opcl), (int, Dev_t, int, int) );
04222     void _PROTOTYPE ((*dmap_io), (int, message *) );
04223     int dmap_driver;
04224     int dmap_flags;
04225 } dmap[];
04226
04227 /*=====
04228 *                         Major and minor device numbers
04229 *=====
04230
04231 /* Total number of different devices. */
04232 #define NR_DEVICES           32             /* number of (major) devices */
04233
04234 /* Major and minor device numbers for MEMORY driver. */
04235 #define MEMORY_MAJOR          1      /* major device for memory devices */
04236 # define RAM_DEV            0      /* minor device for /dev/ram */
04237 # define MEM_DEV            1      /* minor device for /dev/mem */
04238 # define KMEM_DEV           2      /* minor device for /dev/kmem */
04239 # define NULL_DEV           3      /* minor device for /dev/null */
04240 # define BOOT_DEV           4      /* minor device for /dev/boot */
04241 # define ZERO_DEV           5      /* minor device for /dev/zero */
04242
04243 #define CTRLR(n) ((n)==0 ? 3 : (8 + 2*((n)-1))) /* magic formula */
04244
04245 /* Full device numbers that are special to the boot monitor and FS. */
04246 # define DEV_RAM            0x0100 /* device number of /dev/ram */
04247 # define DEV_BOOT           0x0104 /* device number of /dev/boot */
04248
04249 #define FLOPPY_MAJOR          2      /* major device for floppy disks */
04250 #define TTY_MAJOR             4      /* major device for ttys */
04251 #define CTTY_MAJOR            5      /* major device for /dev/tty */
04252
04253 #define INET_MAJOR           7      /* major device for inet */
04254
04255 #define LOG_MAJOR            15     /* major device for log driver */
04256 # define IS_KLOG_DEV        0      /* minor device for /dev/klog */
04257
04258 #endif /* _DMAP_H */

```

```

+++++
include.ibm/portio.h
+++++
04300 /*
04301 ibm/portio.h
04302
04303 Created:      Jan 15, 1992 by Philip Homburg
04304 */
04305
04306 #ifndef _PORTIO_H_
04307 #define _PORTIO_H_
04308
04309 #ifndef _TYPES_H_
04310 #include <sys/types.h>
04311 #endif
04312
04313 unsigned inb(U16_t _port);
04314 unsigned inw(U16_t _port);
04315 unsigned inl(U32_t _port);
04316 void outb(U16_t _port, U8_t _value);
04317 void outw(U16_t _port, U16_t _value);

```

```

04318 void outl(U16_t _port, U32_t _value);
04319 void insb(U16_t _port, void *_buf, size_t _count);
04320 void insw(U16_t _port, void *_buf, size_t _count);
04321 void insl(U16_t _port, void *_buf, size_t _count);
04322 void outsb(U16_t _port, void *_buf, size_t _count);
04323 void outsw(U16_t _port, void *_buf, size_t _count);
04324 void outsl(U16_t _port, void *_buf, size_t _count);
04325 void intr_disable(void);
04326 void intr_enable(void);
04327
04328 #endif /* _PORTIO_H_ */

```

---

```

+++++include/ibm/interrupt.h
+++++
04400 /* Interrupt numbers and hardware vectors. */
04401
04402 #ifndef _INTERRUPT_H
04403 #define _INTERRUPT_H
04404
04405 #if (CHIP == INTEL)
04406
04407 /* 8259A interrupt controller ports. */
04408 #define INT_CTL          0x20    /* I/O port for interrupt controller */
04409 #define INT_CTMASK        0x21    /* setting bits in this port disables ints */
04410 #define INT2_CTL          0xA0    /* I/O port for second interrupt controller */
04411 #define INT2_CTMASK        0xA1    /* setting bits in this port disables ints */
04412
04413 /* Magic numbers for interrupt controller. */
04414 #define END_OF_INT        0x20    /* code used to re-enable after an interrupt */
04415
04416 /* Interrupt vectors defined/reserved by processor. */
04417 #define DIVIDE_VECTOR      0      /* divide error */
04418 #define DEBUG_VECTOR       1      /* single step (trace) */
04419 #define NMI_VECTOR         2      /* non-maskable interrupt */
04420 #define BREAKPOINT_VECTOR  3      /* software breakpoint */
04421 #define OVERFLOW_VECTOR     4      /* from INTO */
04422
04423 /* Fixed system call vector. */
04424 #define SYS_VECTOR          32    /* system calls are made with int SYSVEC */
04425 #define SYS386_VECTOR        33    /* except 386 system calls use this */
04426 #define LEVEL0_VECTOR        34    /* for execution of a function at level 0 */
04427
04428 /* Suitable irq bases for hardware interrupts. Reprogram the 8259(s) from
04429 * the PC BIOS defaults since the BIOS doesn't respect all the processor's
04430 * reserved vectors (0 to 31).
04431 */
04432 #define BIOS IRQ0_VEC      0x08    /* base of IRQ0-7 vectors used by BIOS */
04433 #define BIOS IRQ8_VEC      0x70    /* base of IRQ8-15 vectors used by BIOS */
04434 #define IRQ0_VECTOR         0x50    /* nice vectors to relocate IRQ0-7 to */
04435 #define IRQ8_VECTOR         0x70    /* no need to move IRQ8-15 */
04436
04437 /* Hardware interrupt numbers. */
04438 #define NR_IRQ_VECTORS     16
04439 #define CLOCK_IRQ           0
04440 #define KEYBOARD_IRQ        1
04441 #define CASCADE_IRQ          2    /* cascade enable for 2nd AT controller */
04442 #define ETHER_IRQ            3    /* default ethernet interrupt vector */
04443 #define SECONDARY_IRQ        3    /* RS232 interrupt vector for port 2 */
04444 #define RS232_IRQ             4    /* RS232 interrupt vector for port 1 */
04445 #define XT_WINI_IRQ          5    /* xt winchester */
04446 #define FLOPPY_IRQ            6    /* floppy disk */
04447 #define PRINTER_IRQ          7
04448 #define AT_WINI_0_IRQ         14   /* at winchester controller 0 */
04449 #define AT_WINI_1_IRQ         15   /* at winchester controller 1 */

```

```

04450
04451 /* Interrupt number to hardware vector. */
04452 #define BIOS_VECTOR(irq) \
04453   (((irq) < 8 ? BIOS IRQ0_VEC : BIOS IRQ8_VEC) + ((irq) & 0x07))
04454 #define VECTOR(irq) \
04455   (((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07))
04456
04457 #endif /* (CHIP == INTEL) */
04458
04459 #endif /* _INTERRUPT_H */

+++++
include/ibm/ports.h
+++++

04500 /* Addresses and magic numbers for miscellaneous ports. */
04501
04502 #ifndef _PORTS_H
04503 #define _PORTS_H
04504
04505 #if (CHIP == INTEL)
04506
04507 /* Miscellaneous ports. */
04508 #define PCR          0x65    /* Planar Control Register */
04509 #define PORT_B        0x61    /* I/O port for 8255 port B (kbd, beeper...) */
04510 #define TIMERO        0x40    /* I/O port for timer channel 0 */
04511 #define TIMER2        0x42    /* I/O port for timer channel 2 */
04512 #define TIMER_MODE    0x43    /* I/O port for timer mode control */
04513
04514 #endif /* (CHIP == INTEL) */
04515
04516 #endif /* _PORTS_H */

+++++
kernel/kernel.h
+++++

04600 #ifndef KERNEL_H
04601 #define KERNEL_H
04602
04603 /* This is the master header for the kernel. It includes some other files
 * and defines the principal constants.
04605 */
04606 #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
04607 #define _MINIX             1    /* tell headers to include MINIX stuff */
04608 #define _SYSTEM            1    /* tell headers that this is the kernel */
04609
04610 /* The following are so basic, all the *.c files get them automatically. */
04611 #include <minix/config.h>      /* global configuration, MUST be first */
04612 #include <ansi.h>           /* C style: ANSI 'or K&R, MUST be second */
04613 #include <sys/types.h>        /* general system types */
04614 #include <minix/const.h>       /* MINIX specific constants */
04615 #include <minix/type.h>        /* MINIX specific types, e.g. message */
04616 #include <minix/ipc.h>         /* MINIX run-time system */
04617 #include <timers.h>           /* watchdog timer management */
04618 #include <errno.h>            /* return codes and error numbers */
04619 #include <ibm/portio.h>        /* device I/O and toggle interrupts */
04620
04621 /* Important kernel header files. */
04622 #include "config.h"           /* configuration, MUST be first */
04623 #include "const.h"            /* constants, MUST be second */
04624 #include "type.h"             /* type definitions, MUST be third */
04625 #include "proto.h"            /* function prototypes */
04626 #include "glo.h"              /* global variables */

```

```

04627 #include "ipc.h"           /* IPC constants */
04628 /* #include "debug.h" */    /* debugging, MUST be last kernel header */
04629
04630 #endif /* KERNEL_H */
04631

+++++
kernel/config.h
+++++

04700 #ifndef CONFIG_H
04701 #define CONFIG_H
04702
04703 /* This file defines the kernel configuration. It allows to set sizes of some
04704 * kernel buffers and to enable or disable debugging code, timing features,
04705 * and individual kernel calls.
04706 *
04707 * Changes:
04708 *   Jul 11, 2005      Created. (Jorrit N. Herder)
04709 */
04710
04711 /* In embedded and sensor applications, not all the kernel calls may be
04712 * needed. In this section you can specify which kernel calls are needed
04713 * and which are not. The code for unneeded kernel calls is not included in
04714 * the system binary, making it smaller. If you are not sure, it is best
04715 * to keep all kernel calls enabled.
04716 */
04717 #define USE_FORK          1 /* fork a new process */
04718 #define USE_NEWMAP         1 /* set a new memory map */
04719 #define USE_EXEC           1 /* update process after execute */
04720 #define USE_EXIT           1 /* clean up after process exit */
04721 #define USE_TRACE          1 /* process information and tracing */
04722 #define USE_GETKSIG         1 /* retrieve pending kernel signals */
04723 #define USE_ENDKSIG         1 /* finish pending kernel signals */
04724 #define USE_KILL            1 /* send a signal to a process */
04725 #define USE_SIGSEND         1 /* send POSIX-style signal */
04726 #define USE_SIGRETURN        1 /* sys_sigreturn(proc_nr, ctxt_ptr, flags) */
04727 #define USE_ABORT            1 /* shut down MINIX */
04728 #define USE_GETINFO          1 /* retrieve a copy of kernel data */
04729 #define USE_TIMES             1 /* get process and system time info */
04730 #define USE_SETALARM         1 /* schedule a synchronous alarm */
04731 #define USE_DEVIO            1 /* read or write a single I/O port */
04732 #define USE_VDEVIO           1 /* process vector with I/O requests */
04733 #define USE_SDEVIO           1 /* perform I/O request on a buffer */
04734 #define USE_IRQCTL           1 /* set an interrupt policy */
04735 #define USE_SEGCTL           1 /* set up a remote segment */
04736 #define USE_PRIVCTL          1 /* system privileges control */
04737 #define USE_NICE              1 /* change scheduling priority */
04738 #define USE_UMAP              1 /* map virtual to physical address */
04739 #define USE_VIRCOPY           1 /* copy using virtual addressing */
04740 #define USE_VIRVCOPY          1 /* vector with virtual copy requests */
04741 #define USE_PHYSCOPY          1 /* copy using physical addressing */
04742 #define USE_PHYSVCOPY         1 /* vector with physical copy requests */
04743 #define USE_MEMSET            1 /* write char to a given memory area */
04744
04745 /* Length of program names stored in the process table. This is only used
04746 * for the debugging dumps that can be generated with the IS server. The PM
04747 * server keeps its own copy of the program name.
04748 */
04749 #define P_NAME_LEN           8
04750
04751 /* Kernel diagnostics are written to a circular buffer. After each message,
04752 * a system server is notified and a copy of the buffer can be retrieved to
04753 * display the message. The buffers size can safely be reduced.
04754 */

```

```

04755 #define KMESS_BUF_SIZE 256
04756
04757 /* Buffer to gather randomness. This is used to generate a random stream by
04758 * the MEMORY driver when reading from /dev/random.
04759 */
04760 #define RANDOM_ELEMENTS 32
04761
04762 /* This section contains defines for valuable system resources that are used
04763 * by device drivers. The number of elements of the vectors is determined by
04764 * the maximum needed by any given driver. The number of interrupt hooks may
04765 * be incremented on systems with many device drivers.
04766 */
04767 #define NR_IRQ_HOOKS 16 /* number of interrupt hooks */
04768 #define VDEVIO_BUF_SIZE 64 /* max elements per VDEVIO request */
04769 #define VCOPY_VEC_SIZE 16 /* max elements per VCOPY request */
04770
04771 /* How many bytes for the kernel stack. Space allocated in mpx.s. */
04772 #define K_STACK_BYTES 1024
04773
04774 /* This section allows to enable kernel debugging and timing functionality.
04775 * For normal operation all options should be disabled.
04776 */
04777 #define DEBUG_SCHED_CHECK 0 /* sanity check of scheduling queues */
04778 #define DEBUG_LOCK_CHECK 0 /* kernel lock() sanity check */
04779 #define DEBUG_TIME_LOCKS 0 /* measure time spent in locks */
04780
04781 #endif /* CONFIG_H */
04782

```

```

+++++
kernel/const.h
+++++
04800 /* General macros and constants used by the kernel. */
04801 #ifndef CONST_H
04802 #define CONST_H
04803
04804 #include <ibm/interrupt.h> /* interrupt numbers and hardware vectors */
04805 #include <ibm/ports.h> /* port addresses and magic numbers */
04806 #include <ibm/bios.h> /* BIOS addresses, sizes and magic numbers */
04807 #include <ibm/cpu.h> /* BIOS addresses, sizes and magic numbers */
04808 #include <minix/config.h>
04809 #include "config.h"
04810
04811 /* To translate an address in kernel space to a physical address. This is
04812 * the same as umap_local(proc_ptr, D, vir, sizeof(*vir)), but less costly.
04813 */
04814 #define vir2phys(vir) (kinfo.data_base + (vir_bytes) (vir))
04815
04816 /* Map a process number to a privilege structure id. */
04817 #define s_nr_to_id(n) (NR_TASKS + (n) + 1)
04818
04819 /* Translate a pointer to a field in a structure to a pointer to the structure
04820 * itself. So it translates '&struct_ptr->field' back to 'struct_ptr'.
04821 */
04822 #define structof(type, field, ptr) \
04823     ((type *) (((char *) (ptr)) - offsetof(type, field)))
04824
04825 /* Constants used in virtual_copy(). Values must be 0 and 1, respectively. */
04826 #define _SRC_ 0
04827 #define _DST_ 1
04828
04829 /* Number of random sources */
04830 #define RANDOM_SOURCES 16
04831

```

```

04832 /* Constants and macros for bit map manipulation. */
04833 #define BITCHUNK_BITS  (sizeof(bitchunk_t) * CHAR_BIT)
04834 #define BITMAP_CHUNKS(nr_bits) (((nr_bits)+BITCHUNK_BITS-1)/BITCHUNK_BITS)
04835 #define MAP_CHUNK(map,bit) (map)[((bit)/BITCHUNK_BITS)]
04836 #define CHUNK_OFFSET(bit) ((bit)%BITCHUNK_BITS)
04837 #define GET_BIT(map,bit) ( MAP_CHUNK(map,bit) & (1 << CHUNK_OFFSET(bit)) )
04838 #define SET_BIT(map,bit) ( MAP_CHUNK(map,bit) |= (1 << CHUNK_OFFSET(bit)) )
04839 #define UNSET_BIT(map,bit) ( MAP_CHUNK(map,bit) &= ~(1 << CHUNK_OFFSET(bit)) )
04840
04841 #define get_sys_bit(map,bit) \
04842     ( MAP_CHUNK(map.chunk,bit) & (1 << CHUNK_OFFSET(bit)) )
04843 #define set_sys_bit(map,bit) \
04844     ( MAP_CHUNK(map.chunk,bit) |= (1 << CHUNK_OFFSET(bit)) )
04845 #define unset_sys_bit(map,bit) \
04846     ( MAP_CHUNK(map.chunk,bit) &= ~(1 << CHUNK_OFFSET(bit)) )
04847 #define NR_SYS_CHUNKS  BITMAP_CHUNKS(NR_SYS_PROCS)
04848
04849 /* Program stack words and masks. */
04850 #define INIT_PSW    0x0200 /* initial psw */
04851 #define INIT_TASK_PSW 0x1200 /* initial psw for tasks (with IOPL 1) */
04852 #define TRACEBIT    0x0100 /* OR this with psw in proc[] for tracing */
04853 #define SETPSW(rp,new) /* permits only certain bits to be set */ \
04854     ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xCDS | (new) & 0xCD5)
04855 #define IF_MASK 0x00000200
04856 #define IOPL_MASK 0x0003000
04857
04858 /* Disable/ enable hardware interrupts. The parameters of lock() and unlock()
04859 * are used when debugging is enabled. See debug.h for more information.
04860 */
04861 #define lock(c, v)      intr_disable();
04862 #define unlock(c)       intr_enable();
04863
04864 /* Sizes of memory tables. The boot monitor distinguishes three memory areas,
04865 * namely low mem below 1M, 1M-16M, and mem after 16M. More chunks are needed
04866 * for DOS MINIX.
04867 */
04868 #define NR_MEMS          8
04869
04870 #endif /* CONST_H */
04871
04872
04873
04874
04875

```

```
+++++
kernel/type.h
+++++
```

```

04900 #ifndef TYPE_H
04901 #define TYPE_H
04902
04903 typedef _PROTOTYPE( void task_t, (void) );
04904
04905 /* Process table and system property related types. */
04906 typedef int proc_nr_t;           /* process table entry number */
04907 typedef short sys_id_t;         /* system process index */
04908 typedef struct {
04909     bitchunk_t chunk[BITMAP_CHUNKS(NR_SYS_PROCS)];
04910 } sys_map_t;
04911
04912 struct boot_image {
04913     proc_nr_t proc_nr;           /* process number to use */
04914     task_t *initial_pc;          /* start function for tasks */

```

```

04915     int flags;                      /* process flags */
04916     unsigned char quantum;          /* quantum (tick count) */
04917     int priority;                  /* scheduling priority */
04918     int stksize;                   /* stack size for tasks */
04919     short trap_mask;              /* allowed system call traps */
04920     bitchunk_t ipc_to;            /* send mask protection */
04921     long call_mask;               /* system call protection */
04922     char proc_name[P_NAME_LEN];   /* name in process table */
04923 };
04924
04925 struct memory {
04926     phys_ticks base;                /* start address of chunk */
04927     phys_ticks size;               /* size of memory chunk */
04928 };
04929
04930 /* The kernel outputs diagnostic messages in a circular buffer. */
04931 struct kmessages {
04932     int km_next;                   /* next index to write */
04933     int km_size;                  /* current size in buffer */
04934     char km_buf[KMESS_BUF_SIZE];  /* buffer for messages */
04935 };
04936
04937 struct randomness {
04938     struct {
04939         int r_next;                  /* next index to write */
04940         int r_size;                 /* number of random elements */
04941         unsigned short r_buf[RANDOM_ELEMENTS]; /* buffer for random info */
04942     } bin[RANDOM_SOURCES];
04943 };
04944
04945 #if (CHIP == INTEL)
04946 typedef unsigned reg_t;           /* machine register */
04947
04948 /* The stack frame layout is determined by the software, but for efficiency
04949 * it is laid out so the assembly code to use it is as simple as possible.
04950 * 80286 protected mode and all real modes use the same frame, built with
04951 * 16-bit registers. Real mode lacks an automatic stack switch, so little
04952 * is lost by using the 286 frame for it. The 386 frame differs only in
04953 * having 32-bit registers and more segment registers. The same names are
04954 * used for the larger registers to avoid differences in the code.
04955 */
04956 struct stackframe_s {           /* proc_ptr points here */
04957 #if _WORD_SIZE == 4
04958     u16_t gs;                     /* last item pushed by save */
04959     u16_t fs;
04960 #endif
04961     u16_t es;                     /* | */
04962     u16_t ds;                     /* | */
04963     reg_t di;                    /* di through cx are not accessed in C */
04964     reg_t si;                    /* order is to match pusha/popa */
04965     reg_t fp;                    /* bp */
04966     reg_t st;                    /* hole for another copy of sp */
04967     reg_t bx;                    /* | */
04968     reg_t dx;                    /* | */
04969     reg_t cx;                    /* | */
04970     reg_t retreg;               /* ax and above are all pushed by save */
04971     reg_t retadr;               /* return address for assembly code save() */
04972     reg_t pc;                    /* ^ last item pushed by interrupt */
04973     reg_t cs;                    /* | */
04974     reg_t psw;                  /* | */
04975     reg_t sp;                    /* | */
04976     reg_t ss;                    /* these are pushed by CPU during interrupt */
04977 };
04978
04979 struct segdesc_s {             /* segment descriptor for protected mode */
04980     u16_t limit_low;
04981     u16_t base_low;

```

```

04982     u8_t base_middle;
04983     u8_t access;           /* |P|DL|1|X|E|R|A| */
04984     u8_t granularity;      /* |G|X|0|A|LIMIT| */
04985     u8_t base_high;
04986 };
04987
04988 typedef unsigned long irq_policy_t;
04989 typedef unsigned long irq_id_t;
04990
04991 typedef struct irq_hook {
04992     struct irq_hook *next;      /* next hook in chain */
04993     int (*handler)(struct irq_hook *); /* interrupt handler */
04994     int irq;                  /* IRQ vector number */
04995     int id;                  /* id of this hook */
04996     int proc_nr;              /* NONE if not in use */
04997     irq_id_t notify_id;       /* id to return on interrupt */
04998     irq_policy_t policy;      /* bit mask for policy */
04999 } irq_hook_t;
05000
05001 typedef int (*irq_handler_t)(struct irq_hook *);
05002
05003 #endif /* (CHIP == INTEL) */
05004
05005 #if (CHIP == M68000)
05006 /* M68000 specific types go here. */
05007#endif /* (CHIP == M68000) */
05008
05009#endif /* TYPE_H */

+++++kernel/proto.h+++++
kernel/proto.h
+++++main.c+++++
05100 /* Function prototypes. */
05101
05102 #ifndef PROTO_H
05103 #define PROTO_H
05104
05105 /* Struct declarations. */
05106 struct proc;
05107 struct timer;
05108
05109 /* clock.c */
05110 _PROTOTYPE( void clock_task, (void) );           );
05111 _PROTOTYPE( void clock_stop, (void) );           );
05112 _PROTOTYPE( clock_t get_uptime, (void) );         );
05113 _PROTOTYPE( unsigned long read_clock, (void) );   );
05114 _PROTOTYPE( void set_timer, (struct timer *tp, clock_t t, tmr_func_t f) ); );
05115 _PROTOTYPE( void reset_timer, (struct timer *tp) ); );
05116
05117 /* main.c */
05118 _PROTOTYPE( void main, (void) );                 );
05119 _PROTOTYPE( void prepare_shutdown, (int how) );   );
05120
05121 /* utility.c */
05122 _PROTOTYPE( void kprintf, (const char *fmt, ...) ); );
05123 _PROTOTYPE( void panic, (_CONST char *s, int n) ); );
05124
05125 /* proc.c */
05126 _PROTOTYPE( int sys_call, (int function, int src_dest, message *m_ptr) ); );
05127 _PROTOTYPE( int lock_notify, (int src, int dst) ); );
05128 _PROTOTYPE( int lock_send, (int dst, message *m_ptr) ); );
05129 _PROTOTYPE( void lock_enqueue, (struct proc *rp) ); );
05130 _PROTOTYPE( void lock_dequeue, (struct proc *rp) ); );
05131
05132 /* start.c */
05133 _PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mds,

```

```

05134             U16_t parmoff, U16_t parmsize) );
05135
05136 /* system.c */
05137 _PROTOTYPE( int get_priv, (register struct proc *rc, int proc_type) );
05138 _PROTOTYPE( void send_sig, (int proc_nr, int sig_nr) );
05139 _PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr) );
05140 _PROTOTYPE( void sys_task, (void) );
05141 _PROTOTYPE( void get_randomness, (int source) );
05142 _PROTOTYPE( int virtual_copy, (struct vir_addr *src, struct vir_addr *dst,
05143                           vir_bytes bytes) );
05144 #define numa_local(proc_nr, vir_addr, bytes) \
05145     umap_local(proc_addr(proc_nr), D, (vir_addr), (bytes))
05146 _PROTOTYPE( phys_bytes umap_local, (struct proc *rp, int seg,
05147               vir_bytes vir_addr, vir_bytes bytes) );
05148 _PROTOTYPE( phys_bytes umap_remote, (struct proc *rp, int seg,
05149               vir_bytes vir_addr, vir_bytes bytes) );
05150 _PROTOTYPE( phys_bytes umap_bios, (struct proc *rp, vir_bytes vir_addr,
05151               vir_bytes bytes) );
05152
05153 /* exception.c */
05154 _PROTOTYPE( void exception, (unsigned vec_nr) );
05155
05156 /* i8259.c */
05157 _PROTOTYPE( void intr_init, (int mine) );
05158 _PROTOTYPE( void intr_handle, (irq_hook_t *hook) );
05159 _PROTOTYPE( void put_irq_handler, (irq_hook_t *hook, int irq,
05160                           irq_handler_t handler) );
05161 _PROTOTYPE( void rm_irq_handler, (irq_hook_t *hook) );
05162
05163 /* klib*.s */
05164 _PROTOTYPE( void int86, (void) );
05165 _PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
05166               phys_clicks dst_clicks, vir_bytes dst_offset) );
05167 _PROTOTYPE( void enable_irq, (irq_hook_t *hook) );
05168 _PROTOTYPE( void disable_irq, (irq_hook_t *hook) );
05169 _PROTOTYPE( u16_t mem_rdw, (U16_t segm, vir_bytes offset) );
05170 _PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
05171               phys_bytes count) );
05172 _PROTOTYPE( void phys_memset, (phys_bytes source, unsigned long pattern,
05173               phys_bytes count) );
05174 _PROTOTYPE( void phys_insb, (U16_t port, phys_bytes buf, size_t count) );
05175 _PROTOTYPE( void phys_insw, (U16_t port, phys_bytes buf, size_t count) );
05176 _PROTOTYPE( void phys_outsb, (U16_t port, phys_bytes buf, size_t count) );
05177 _PROTOTYPE( void phys_outsw, (U16_t port, phys_bytes buf, size_t count) );
05178 _PROTOTYPE( void reset, (void) );
05179 _PROTOTYPE( void level0, (void (*func)(void)) );
05180 _PROTOTYPE( void monitor, (void) );
05181 _PROTOTYPE( void read_tsc, (unsigned long *high, unsigned long *low) );
05182 _PROTOTYPE( unsigned long read_cpu_flags, (void) );
05183
05184 /* mpx*.s */
05185 _PROTOTYPE( void idle_task, (void) );
05186 _PROTOTYPE( void restart, (void) );
05187
05188 /* The following are never called from C (pure asm procs). */
05189
05190 /* Exception handlers (real or protected mode), in numerical order. */
05191 void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
05192 void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
05193 void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
05194 void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
05195 void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
05196 void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
05197 void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
05198 void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
05199 void _PROTOTYPE( copr_double_fault, (void) );
05200 void _PROTOTYPE( copr_seg_overrun, (void) );

```

```

05201 void _PROTOTYPE( inval_tss, (void) );
05202 void _PROTOTYPE( segment_not_present, (void) );
05203 void _PROTOTYPE( stack_exception, (void) );
05204 void _PROTOTYPE( general_protection, (void) );
05205 void _PROTOTYPE( page_fault, (void) );
05206 void _PROTOTYPE( copr_error, (void) );
05207
05208 /* Hardware interrupt handlers. */
05209 _PROTOTYPE( void hwind00, (void) );
05210 _PROTOTYPE( void hwind01, (void) );
05211 _PROTOTYPE( void hwind02, (void) );
05212 _PROTOTYPE( void hwind03, (void) );
05213 _PROTOTYPE( void hwind04, (void) );
05214 _PROTOTYPE( void hwind05, (void) );
05215 _PROTOTYPE( void hwind06, (void) );
05216 _PROTOTYPE( void hwind07, (void) );
05217 _PROTOTYPE( void hwind08, (void) );
05218 _PROTOTYPE( void hwind09, (void) );
05219 _PROTOTYPE( void hwind10, (void) );
05220 _PROTOTYPE( void hwind11, (void) );
05221 _PROTOTYPE( void hwind12, (void) );
05222 _PROTOTYPE( void hwind13, (void) );
05223 _PROTOTYPE( void hwind14, (void) );
05224 _PROTOTYPE( void hwind15, (void) );
05225
05226 /* Software interrupt handlers, in numerical order. */
05227 _PROTOTYPE( void trp, (void) );
05228 _PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
05229 _PROTOTYPE( void level0_call, (void) );
05230
05231 /* protect.c */
05232 _PROTOTYPE( void prot_init, (void) );
05233 _PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
05234 vir_bytes size, int privilege) );
05235 _PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
05236 vir_bytes size, int privilege) );
05237 _PROTOTYPE( phys_bytes seg2phys, (U16_t seg) );
05238 _PROTOTYPE( void phys2seg, (u16_t *seg, vir_bytes *off, phys_bytes phys));
05239 _PROTOTYPE( void enable_iop, (struct proc *pp) );
05240 _PROTOTYPE( void alloc_segments, (struct proc *rp) );
05241
05242 #endif /* PROTO_H */
05243
05244

```

```
+++++
kernel/glo.h
+++++
```

```

05300 #ifndef GLO_H
05301 #define GLO_H
05302
05303 /* Global variables used in the kernel. This file contains the declarations;
05304 * storage space for the variables is allocated in table.c, because EXTERN is
05305 * defined as extern unless the _TABLE definition is seen. We rely on the
05306 * compiler's default initialization (0) for several global variables.
05307 */
05308 #ifdef _TABLE
05309 #undef EXTERN
05310 #define EXTERN
05311 #endif
05312
05313 #include <minix/config.h>
05314 #include "config.h"
05315
05316 /* Variables relating to shutting down MINIX. */
05317 EXTERN char kernel_exception; /* TRUE after system exceptions */
```

```

05318 EXTERN char shutdown_started;           /* TRUE after shutdowns / reboots */
05319
05320 /* Kernel information structures. This groups vital kernel information. */
05321 EXTERN phys_bytes aout;                /* address of a.out headers */
05322 EXTERN struct kinfo kinfo;             /* kernel information for users */
05323 EXTERN struct machine machine;        /* machine information for users */
05324 EXTERN struct kmessages kmess;         /* diagnostic messages in kernel */
05325 EXTERN struct randomness krandom;     /* gather kernel random information */
05326
05327 /* Process scheduling information and the kernel reentry count. */
05328 EXTERN struct proc *prev_ptr;          /* previously running process */
05329 EXTERN struct proc *proc_ptr;          /* pointer to currently running process */
05330 EXTERN struct proc *next_ptr;          /* next process to run after restart() */
05331 EXTERN struct proc *bill_ptr;          /* process to bill for clock ticks */
05332 EXTERN char k_reenter;                /* kernel reentry count (entry count less 1) */
05333 EXTERN unsigned lost_ticks;           /* clock ticks counted outside clock task */
05334
05335 /* Interrupt related variables. */
05336 EXTERN irq_hook_t irq_hooks[NR_IRQ_HOOKS]; /* hooks for general use */
05337 EXTERN irq_hook_t *irq_handlers[NR_IRQ_VECTORS]; /* list of IRQ handlers */
05338 EXTERN int irq_actids[NR_IRQ_VECTORS];      /* IRQ ID bits active */
05339 EXTERN int irq_use;                     /* map of all in-use irq's */
05340
05341 /* Miscellaneous. */
05342 EXTERN reg_t mon_ss, mon_sp;            /* boot monitor stack */
05343 EXTERN int mon_return;                 /* true if we can return to monitor */
05344
05345 /* Variables that are initialized elsewhere are just extern here. */
05346 extern struct boot_image image[];        /* system image processes */
05347 extern char *t_stack[];                  /* task stack space */
05348 extern struct segdesc_s gdt[];           /* global descriptor table */
05349
05350 EXTERN _PROTOTYPE( void (*level0_func), (void) );
05351
05352 #endif /* GLO_H */
05353
05354
05355
05356
05357

```

```
*****
kernel/ipc.h
*****
```

```

05400 #ifndef IPC_H
05401 #define IPC_H
05402
05403 /* This header file defines constants for MINIX inter-process communication.
05404 * These definitions are used in the file proc.c.
05405 */
05406 #include <minix/com.h>
05407
05408 /* Masks and flags for system calls. */
05409 #define SYSCALL_FUNC 0x0F    /* mask for system call function */
05410 #define SYSCALL_FLAGS 0xF0   /* mask for system call flags */
05411 #define NON_BLOCKING 0x10   /* prevent blocking, return error */
05412
05413 /* System call numbers that are passed when trapping to the kernel. The
05414 * numbers are carefully defined so that it can easily be seen (based on
05415 * the bits that are on) which checks should be done in sys_call().
05416 */
05417 #define SEND          1    /* 0 0 0 1 : blocking send */
05418 #define RECEIVE       2    /* 0 0 1 0 : blocking receive */
05419 #define SENDREC       3    /* 0 0 1 1 : SEND + RECEIVE */

```

```

05420 #define NOTIFY          4 /* 0 1 0 0 : nonblocking notify */
05421 #define ECHO            8 /* 1 0 0 0 : echo a message */
05422
05423 /* The following bit masks determine what checks that should be done. */
05424 #define CHECK_PTR        0x0B /* 1 0 1 1 : validate message buffer */
05425 #define CHECK_DST         0x05 /* 0 1 0 1 : validate message destination */
05426 #define CHECK_SRC         0x02 /* 0 0 1 0 : validate message source */
05427
05428 #endif /* IPC_H */

```

---

```

+++++kernel/proc.h+++++

```

```

05500 #ifndef PROC_H
05501 #define PROC_H
05502
05503 /* Here is the declaration of the process table. It contains all process
05504 * data, including registers, flags, scheduling priority, memory map,
05505 * accounting, message passing (IPC) information, and so on.
05506 *
05507 * Many assembly code routines reference fields in it. The offsets to these
05508 * fields are defined in the assembler include file sconst.h. When changing
05509 * struct proc, be sure to change sconst.h to match.
05510 */
05511 #include <minix/com.h>
05512 #include "protect.h"
05513 #include "const.h"
05514 #include "priv.h"
05515
05516 struct proc {
05517     struct stackframe_s p_reg;    /* process' registers saved in stack frame */
05518     reg_t p_ldt_sel;           /* selector in gdt with ldt base and limit */
05519     struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments */
05520
05521     proc_nr_t p_nr;           /* number of this process (for fast access) */
05522     struct priv *p_priv;       /* system privileges structure */
05523     char p_rts_flags;         /* SENDING, RECEIVING, etc. */
05524
05525     char p_priority;          /* current scheduling priority */
05526     char p_max_priority;      /* maximum scheduling priority */
05527     char p_ticks_left;        /* number of scheduling ticks left */
05528     char p_quantum_size;      /* quantum size in ticks */
05529
05530     struct mem_map p_memmap[NR_LOCAL_SEGS]; /* memory map (T, D, S) */
05531
05532     clock_t p_user_time;      /* user time in ticks */
05533     clock_t p_sys_time;       /* sys time in ticks */
05534
05535     struct proc *p_nextready;  /* pointer to next ready process */
05536     struct proc *p_caller_q;   /* head of list of procs wishing to send */
05537     struct proc *p_q_link;     /* link to next proc wishing to send */
05538     message *p_messbuf;       /* pointer to passed message buffer */
05539     proc_nr_t p_getfrom;      /* from whom does process want to receive? */
05540     proc_nr_t p_sendto;       /* to whom does process want to send? */
05541
05542     sigset_t p_pending;       /* bit map for pending kernel signals */
05543
05544     char p_name[P_NAME_LEN];  /* name of the process, including \0 */
05545 };
05546
05547 /* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */
05548 #define SLOT_FREE          0x01 /* process slot is free */
05549 #define NO_MAP              0x02 /* keeps unmapped forked child from running */
05550 #define SENDING             0x04 /* process blocked trying to SEND */
05551 #define RECEIVING            0x08 /* process blocked trying to RECEIVE */
05552 #define SIGNALED             0x10 /* set when new kernel signal arrives */

```

```

05553 #define SIG_PENDING      0x20    /* unready while signal being processed */
05554 #define P_STOP          0x40    /* set when process is being traced */
05555 #define NO_PRIV         0x80    /* keep forked system process from running */
05556
05557 /* Scheduling priorities for p_priority. Values must start at zero (highest
   * priority) and increment. Priorities of the processes in the boot image
   * can be set in table.c. IDLE must have a queue for itself, to prevent low
   * priority user processes to run round-robin with IDLE.
   */
05562 #define NR_SCHED_QUEUES 16     /* MUST equal minimum priority + 1 */
05563 #define TASK_Q           0       /* highest, used for kernel tasks */
05564 #define MAX_USER_Q        0       /* highest priority for user processes */
05565 #define USER_Q            7       /* default (should correspond to nice 0) */
05566 #define MIN_USER_Q        14      /* minimum priority for user processes */
05567 #define IDLE_Q            15      /* lowest, only IDLE process goes here */
05568
05569 /* Magic process table addresses. */
05570 #define BEG_PROC_ADDR (&proc[0])
05571 #define BEG_USER_ADDR (&proc[NR_TASKS])
05572 #define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
05573
05574 #define NIL_PROC          ((struct proc *) 0)
05575 #define NIL_SYS_PROC       ((struct proc *) 1)
05576 #define cproc_addr(n)      (&(proc + NR_TASKS)[(n)])
05577 #define proc_addr(n)       (pproc_addr + NR_TASKS)[(n)]
05578 #define proc_nr(p)         ((p)->p_nr)
05579
05580 #define isokprocn(n)      (((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS))
05581 #define isemptyn(n)        isemptyp(proc_addr(n))
05582 #define isemptyp(p)        ((p)->p_rts_flags == SLOT_FREE)
05583 #define iskernelp(p)       iskerneln((p)->p_nr)
05584 #define iskerneln(n)       ((n) < 0)
05585 #define isusersp(p)        isusern((p)->p_nr)
05586 #define isusern(n)         ((n) >= 0)
05587
05588 /* The process table and pointers to process table slots. The pointers allow
   * faster access because now a process entry can be found by indexing the
   * pproc_addr array, while accessing an element i requires a multiplication
   * with sizeof(struct proc) to determine the address.
   */
05589 EXTERN struct proc proc[NR_TASKS + NR_PROCS]; /* process table */
05590 EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
05591 EXTERN struct proc *rdy_head[NR_SCHED_QUEUES]; /* ptrs to ready list headers */
05592 EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES]; /* ptrs to ready list tails */
05593
05598 #endif /* PROC_H */

```

+++++  
kernel/sconst.h  
+++++

```

05600 ! Miscellaneous constants used in assembler code.
05601 W           = _WORD_SIZE      ! Machine word size.
05602
05603 ! Offsets in struct proc. They MUST match proc.h.
05604 P_STACKBASE   = 0
05605 GSREG        = P_STACKBASE
05606 FSREG        = GSREG + 2    ! 386 introduces FS and GS segments
05607 ESREG        = FSREG + 2
05608 DSREG        = ESREG + 2
05609 DIREG        = DSREG + 2
05610 SIREG        = DIREG + W
05611 BPREG        = SIREG + W
05612 STREG        = BPREG + W    ! hole for another SP
05613 BXREG        = STREG + W
05614 DXREG        = BXREG + W

```

```

05615 CXREG      =      DXREG + W
05616 AXREG      =      CXREG + W
05617 RETADR     =      AXREG + W      ! return address for save() call
05618 PCREG      =      RETADR + W
05619 CSREG      =      PCREG + W
05620 PSWREG     =      CSREG + W
05621 SPREG      =      PSWREG + W
05622 SSREG      =      SPREG + W
05623 P_STACKTOP =      SSREG + W
05624 P_LDT_SEL   =      P_STACKTOP
05625 P_LDT      =      P_LDT_SEL + W
05626
05627 Msize       =      9           ! size of a message in 32-bit words

```

```
***** kernel/priv.h *****
```

```

05700 #ifndef PRIV_H
05701 #define PRIV_H
05702
05703 /* Declaration of the system privileges structure. It defines flags, system
05704 * call masks, an synchronous alarm timer, I/O privileges, pending hardware
05705 * interrupts and notifications, and so on.
05706 * System processes each get their own structure with properties, whereas all
05707 * user processes share one structure. This setup provides a clear separation
05708 * between common and privileged process fields and is very space efficient.
05709 *
05710 * Changes:
05711 *   Jul 01, 2005      Created. (Jorrit N. Herder)
05712 */
05713 #include <minix/com.h>
05714 #include "protect.h"
05715 #include "const.h"
05716 #include "type.h"
05717
05718 struct priv {
05719     proc_nr_t s_proc_nr;          /* number of associated process */
05720     sys_id_t s_id;               /* index of this system structure */
05721     short s_flags;              /* PREEMPTIBLE, BILLABLE, etc. */
05722
05723     short s_trap_mask;           /* allowed system call traps */
05724     sys_map_t s_ipc_from;        /* allowed callers to receive from */
05725     sys_map_t s_ipc_to;          /* allowed destination processes */
05726     long s_call_mask;            /* allowed kernel calls */
05727
05728     sys_map_t s_notify_pending;  /* bit map with pending notifications */
05729     irq_id_t s_int_pending;      /* pending hardware interrupts */
05730     sigset_t s_sig_pending;      /* pending signals */
05731
05732     timer_t s_alarm_timer;       /* synchronous alarm timer */
05733     struct far_mem s_farmem[NR_REMOTE_SEGS]; /* remote memory map */
05734     reg_t *s_stack_guard;        /* stack guard word for kernel tasks */
05735 };
05736
05737 /* Guard word for task stacks. */
05738 #define STACK_GUARD ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))
05739
05740 /* Bits for the system property flags. */
05741 #define PREEMPTIBLE 0x01 /* kernel tasks are not preemptible */
05742 #define BILLABLE    0x04 /* some processes are not billable */
05743 #define SYS_PROC    0x10 /* system processes are privileged */
05744 #define SENDREC_BUSY 0x20 /* sendrec() in progress */
05745
05746 /* Magic system structure table addresses. */

```

```

05747 #define BEG_PRIV_ADDR (&priv[0])
05748 #define END_PRIV_ADDR (&priv[NR_SYS_PROCS])
05749
05750 #define priv_addr(i)      (ppriv_addr)[(i)]
05751 #define priv_id(rp)        ((rp)->p_priv->s_id)
05752 #define priv(rp)           ((rp)->p_priv)
05753
05754 #define id_to_nr(id)     priv_addr(id)->s_proc_nr
05755 #define nr_to_id(nr)      priv(proc_addr(nr))->s_id
05756
05757 /* The system structures table and pointers to individual table slots. The
05758 * pointers allow faster access because now a process entry can be found by
05759 * indexing the psys_addr array, while accessing an element i requires a
05760 * multiplication with sizeof(struct sys) to determine the address.
05761 */
05762 EXTERN struct priv priv[NR_SYS_PROCS];          /* system properties table */
05763 EXTERN struct priv *ppriv_addr[NR_SYS_PROCS];    /* direct slot pointers */
05764
05765 /* Unprivileged user processes all share the same privilege structure.
05766 * This id must be fixed because it is used to check send mask entries.
05767 */
05768 #define USER_PRIV_ID     0
05769
05770 /* Make sure the system can boot. The following sanity check verifies that
05771 * the system privileges table is large enough for the number of processes
05772 * in the boot image.
05773 */
05774 #if (NR_BOOT_PROCS > NR_SYS_PROCS)
05775 #error NR_SYS_PROCS must be larger than NR_BOOT_PROCS
05776 #endif
05777
05778 #endif /* PRIV_H */

```

```

+++++
kernel/protect.h
+++++
05800 /* Constants for protected mode. */
05801
05802 /* Table sizes. */
05803 #define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS)
05804                                     /* spec. and LDT's */
05805 #define IDT_SIZE (IRQ8_VECTOR + 8)      /* only up to the highest vector */
05806 #define LDT_SIZE (2 + NR_REMOTE_SEGS)   /* CS, DS and remote segments */
05807
05808 /* Fixed global descriptors. 1 to 7 are prescribed by the BIOS. */
05809 #define GDT_INDEX          1 /* GDT descriptor */
05810 #define IDT_INDEX          2 /* IDT descriptor */
05811 #define DS_INDEX           3 /* kernel DS */
05812 #define ES_INDEX           4 /* kernel ES (386: flag 4 Gb at startup) */
05813 #define SS_INDEX           5 /* kernel SS (386: monitor SS at startup) */
05814 #define CS_INDEX           6 /* kernel CS */
05815 #define MON_CS_INDEX       7 /* temp for BIOS (386: monitor CS at startup) */
05816 #define TSS_INDEX          8 /* kernel TSS */
05817 #define DS_286_INDEX        9 /* scratch 16-bit source segment */
05818 #define ES_286_INDEX        10 /* scratch 16-bit destination segment */
05819 #define A_INDEX             11 /* 64K memory segment at A0000 */
05820 #define B_INDEX             12 /* 64K memory segment at B0000 */
05821 #define C_INDEX             13 /* 64K memory segment at C0000 */
05822 #define D_INDEX             14 /* 64K memory segment at D0000 */
05823 #define FIRST_LDT_INDEX     15 /* rest of descriptors are LDT's */
05824
05825 #define GDT_SELECTOR        0x08 /* (GDT_INDEX * DESC_SIZE) bad for asld */
05826 #define IDT_SELECTOR        0x10 /* (IDT_INDEX * DESC_SIZE) */
05827 #define DS_SELECTOR         0x18 /* (DS_INDEX * DESC_SIZE) */
05828 #define ES_SELECTOR         0x20 /* (ES_INDEX * DESC_SIZE) */

```

```

05829 #define FLAT_DS_SELECTOR 0x21 /* less privileged ES */
05830 #define SS_SELECTOR      0x28 /* (SS_INDEX * DESC_SIZE) */
05831 #define CS_SELECTOR      0x30 /* (CS_INDEX * DESC_SIZE) */
05832 #define MON_CS_SELECTOR 0x38 /* (MON_CS_INDEX * DESC_SIZE) */
05833 #define TSS_SELECTOR     0x40 /* (TSS_INDEX * DESC_SIZE) */
05834 #define DS_286_SELECTOR 0x49 /* (DS_286_INDEX*DESC_SIZE+TASK_PRIVILEGE) */
05835 #define ES_286_SELECTOR 0x51 /* (ES_286_INDEX*DESC_SIZE+TASK_PRIVILEGE) */
05836
05837 /* Fixed local descriptors. */
05838 #define CS_LDT_INDEX      0 /* process CS */
05839 #define DS_LDT_INDEX      1 /* process DS=ES=FS=GS=SS */
05840 #define EXTRA_LDT_INDEX   2 /* first of the extra LDT entries */
05841
05842 /* Privileges. */
05843 #define INTR_PRIVILEGE   0 /* kernel and interrupt handlers */
05844 #define TASK_PRIVILEGE    1 /* kernel tasks */
05845 #define USER_PRIVILEGE    3 /* servers and user processes */
05846
05847 /* 286 hardware constants. */
05848
05849 /* Exception vector numbers. */
05850 #define BOUNDS_VECTOR     5 /* bounds check failed */
05851 #define INVAL_OP_VECTOR   6 /* invalid opcode */
05852 #define COPROC_NOT_VECTOR 7 /* coprocessor not available */
05853 #define DOUBLE_FAULT_VECTOR 8
05854 #define COPROC_SEG_VECTOR  9 /* coprocessor segment overrun */
05855 #define INVAL_TSS_VECTOR  10 /* invalid TSS */
05856 #define SEG_NOT_VECTOR    11 /* segment not present */
05857 #define STACK_FAULT_VECTOR 12 /* stack exception */
05858 #define PROTECTION_VECTOR 13 /* general protection */
05859
05860 /* Selector bits. */
05861 #define TI                0x04 /* table indicator */
05862 #define RPL               0x03 /* requester privilege level */
05863
05864 /* Descriptor structure offsets. */
05865 #define DESC_BASE          2 /* to base_low */
05866 #define DESC_BASE_MIDDLE   4 /* to base_middle */
05867 #define DESC_ACCESS         5 /* to access byte */
05868 #define DESC_SIZE           8 /* sizeof (struct segdesc_s) */
05869
05870 /* Base and limit sizes and shifts. */
05871 #define BASE_MIDDLE_SHIFT 16 /* shift for base --> base_middle */
05872
05873 /* Access-byte and type-byte bits. */
05874 #define PRESENT           0x80 /* set for descriptor present */
05875 #define DPL                0x60 /* descriptor privilege level mask */
05876 #define DESC_SHIFT          5
05877 #define SEGMENT             0x10 /* set for segment-type descriptors */
05878
05879 /* Access-byte bits. */
05880 #define EXECUTABLE        0x08 /* set for executable segment */
05881 #define CONFORMING         0x04 /* set for conforming segment if executable */
05882 #define EXPAND_DOWN        0x04 /* set for expand-down segment if !executable*/
05883 #define READABLE           0x02 /* set for readable segment if executable */
05884 #define WRITEABLE          0x02 /* set for writeable segment if !executable */
05885 #define TSS_BUSY            0x02 /* set if TSS descriptor is busy */
05886 #define ACCESSED           0x01 /* set if segment accessed */
05887
05888 /* Special descriptor types. */
05889 #define AVL_286_TSS        1 /* available 286 TSS */
05890 #define LDT                2 /* local descriptor table */
05891 #define BUSY_286_TSS        3 /* set transparently to the software */
05892 #define CALL_286_GATE       4 /* not used */
05893 #define TASK_GATE          5 /* only used by debugger */
05894 #define INT_286_GATE        6 /* interrupt gate, used for all vectors */
05895 #define TRAP_286_GATE       7 /* not used */

```

```

05896 /* Extra 386 hardware constants. */
05897
05898
05899 /* Exception vector numbers. */
05900 #define PAGE_FAULT_VECTOR    14
05901 #define COPROC_ERR_VECTOR   16 /* coprocessor error */
05902
05903 /* Descriptor structure offsets. */
05904 #define DESC_GRANULARITY     6 /* to granularity byte */
05905 #define DESC_BASE_HIGH       7 /* to base_high */
05906
05907 /* Base and limit sizes and shifts. */
05908 #define BASE_HIGH_SHIFT      24 /* shift for base --> base_high */
05909 #define BYTE_GRAN_MAX        0xFFFFFL /* maximum size for byte granular segment */
05910 #define GRANULARITY_SHIFT    16 /* shift for limit --> granularity */
05911 #define OFFSET_HIGH_SHIFT    16 /* shift for (gate) offset --> offset_high */
05912 #define PAGE_GRAN_SHIFT      12 /* extra shift for page granular limits */
05913
05914 /* Type-byte bits. */
05915 #define DESC_386_BIT        0x08 /* 386 types are obtained by ORing with this */
05916 /*      /* LDT's and TASK_GATE's don't need it */
05917
05918 /* Granularity byte. */
05919 #define GRANULAR          0x80 /* set for 4K granularity */
05920 #define DEFAULT           0x40 /* set for 32-bit defaults (executable seg) */
05921 #define BIG               0x40 /* set for "BIG" (expand-down seg) */
05922 #define AVL              0x10 /* 0 for available */
05923 #define LIMIT_HIGH        0XF /* mask for high bits of limit */

```

```

+++++
kernel/table.c
++++

06000 /* The object file of "table.c" contains most kernel data. Variables that
06001 * are declared in the *.h files appear with EXTERN in front of them, as in
06002 *
06003 *     EXTERN int x;
06004 *
06005 * Normally EXTERN is defined as extern, so when they are included in another
06006 * file, no storage is allocated. If EXTERN were not present, but just say,
06007 *
06008 *     int x;
06009 *
06010 * then including this file in several source files would cause 'x' to be
06011 * declared several times. While some linkers accept this, others do not,
06012 * so they are declared extern when included normally. However, it must be
06013 * declared for real somewhere. That is done here, by redefining EXTERN as
06014 * the null string, so that inclusion of all *.h files in table.c actually
06015 * generates storage for them.
06016 *
06017 * Various variables could not be declared EXTERN, but are declared PUBLIC
06018 * or PRIVATE. The reason for this is that extern variables cannot have a
06019 * default initialization. If such variables are shared, they must also be
06020 * declared in one of the *.h files without the initialization. Examples
06021 * include 'boot_image' (this file) and 'idt' and 'gdt' (protect.c).
06022 *
06023 * Changes:
06024 *     Aug 02, 2005  set privileges and minimal boot image (Jorrit N. Herder)
06025 *     Oct 17, 2004  updated above and tasktab comments (Jorrit N. Herder)
06026 *     May 01, 2004  changed struct for system image (Jorrit N. Herder)
06027 */
06028 #define _TABLE
06029
06030 #include "kernel.h"
06031 #include "proc.h"
06032 #include "ipc.h"

```

```

06033 #include <minix/com.h>
06034 #include <ibm/int86.h>
06035
06036 /* Define stack sizes for the kernel tasks included in the system image. */
06037 #define NO_STACK          0
06038 #define SMALL_STACK      (128 * sizeof(char *))
06039 #define IDL_S    SMALL_STACK /* 3 intr, 3 temps, 4 db for Intel */
06040 #define HRD_S    NO_STACK   /* dummy task, uses kernel stack */
06041 #define TSK_S    SMALL_STACK /* system and clock task */
06042
06043 /* Stack space for all the task stacks. Declared as (char *) to align it. */
06044 #define TOT_STACK_SPACE (IDL_S + HRD_S + (2 * TSK_S))
06045 PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];
06046
06047 /* Define flags for the various process types. */
06048 #define IDL_F    (SYS_PROC | PREEMPTIBLE | BILLABLE)      /* idle task */
06049 #define TSK_F    (SYS_PROC)                                /* kernel tasks */
06050 #define SRV_F    (SYS_PROC | PREEMPTIBLE)                  /* system services */
06051 #define USR_F    (BILLABLE | PREEMPTIBLE)                 /* user processes */
06052
06053 /* Define system call traps for the various process types. These call masks
06054 * determine what system call traps a process is allowed to make.
06055 */
06056 #define TSK_T    (1 << RECEIVE)           /* clock and system */
06057 #define SRV_T    (^0)                   /* system services */
06058 #define USR_T    ((1 << SENDREC) | (1 << ECHO)) /* user processes */
06059
06060 /* Send masks determine to whom processes can send messages or notifications.
06061 * The values here are used for the processes in the boot image. We rely on
06062 * the initialization code in main() to match the s_nr_to_id() mapping for the
06063 * processes in the boot image, so that the send mask that is defined here
06064 * can be directly copied onto map[0] of the actual send mask. Privilege
06065 * structure 0 is shared by user processes.
06066 */
06067 #define s(n)      (1 << s_nr_to_id(n))
06068 #define SRV_M (^0)
06069 #define SYS_M (^0)
06070 #define USR_M (s(PM_PROC_NR) | s(FS_PROC_NR) | s(RS_PROC_NR))
06071 #define DRV_M (USR_M | s(SYSTEM) | s(CLOCK) | s(LOG_PROC_NR) | s(TTY_PROC_NR))
06072
06073 /* Define kernel calls that processes are allowed to make. This is not looking
06074 * very nice, but we need to define the access rights on a per call basis.
06075 * Note that the reincarnation server has all bits on, because it should
06076 * be allowed to distribute rights to services that it starts.
06077 */
06078 #define c(n)      (1 << ((n)-KERNEL_CALL))
06079 #define RS_C      ^0
06080 #define PM_C      ~(c(SYS_DEVIO) | c(SYS_SDEVIO) | c(SYS_VDEVIO) \
06081     | c(SYS_IRQCTL) | c(SYS_INT86))
06082 #define FS_C      (c(SYS_KILL) | c(SYS_VIRCOPY) | c(SYS_VIRVCOPY) | c(SYS_UMAP) \
06083     | c(SYS_GETINFO) | c(SYS_EXIT) | c(SYS_TIMES) | c(SYS_SETALARM))
06084 #define DRV_C      (FS_C | c(SYS_SEGCTL) | c(SYS_IRQCTL) | c(SYS_INT86) \
06085     | c(SYS_DEVIO) | c(SYS_VDEVIO) | c(SYS_SDEVIO))
06086 #define MEM_C      (DRV_C | c(SYS_PHYSCOPY) | c(SYS_PHYSVCOPY))
06087
06088 /* The system image table lists all programs that are part of the boot image.
06089 * The order of the entries here MUST agree with the order of the programs
06090 * in the boot image and all kernel tasks must come first.
06091 * Each entry provides the process number, flags, quantum size (qs), scheduling
06092 * queue, allowed traps, ipc mask, and a name for the process table. The
06093 * initial program counter and stack size is also provided for kernel tasks.
06094 */
06095 PUBLIC struct boot_image image[] = {
06096 /* process nr, pc, flags, qs, queue, stack, traps, ipcto, call, name */
06097 { IDLE, idle_task, IDL_F, 8, IDLE_Q, IDL_S, 0, 0, 0, "IDLE" },
06098 { CLOCK,clock_task, TSK_F, 64, TASK_Q, TSK_S, TSK_T, 0, 0, "CLOCK" },
06099 { SYSTEM,sys_task, TSK_F, 64, TASK_Q, TSK_S, TSK_T, 0, 0, "SYSTEM" },

```

```

06100 { HARDWARE, 0, TSK_F, 64, TASK_Q, HRD_S, 0, 0, 0, "KERNEL"},  

06101 { PM_PROC_NR, 0, SRV_F, 32, 3, 0, SRV_T, SRV_M, PM_C, "pm" },  

06102 { FS_PROC_NR, 0, SRV_F, 32, 4, 0, SRV_T, SRV_M, FS_C, "fs" },  

06103 { RS_PROC_NR, 0, SRV_F, 4, 3, 0, SRV_T, SYS_M, RS_C, "rs" },  

06104 { TTY_PROC_NR, 0, SRV_F, 4, 1, 0, SRV_T, SYS_M, DRV_C, "tty" },  

06105 { MEM_PROC_NR, 0, SRV_F, 4, 2, 0, SRV_T, DRV_M, MEM_C, "memory"},  

06106 { LOG_PROC_NR, 0, SRV_F, 4, 2, 0, SRV_T, SYS_M, DRV_C, "log" },  

06107 { DRVR_PROC_NR, 0, SRV_F, 4, 2, 0, SRV_T, SYS_M, DRV_C, "driver"},  

06108 { INIT_PROC_NR, 0, USR_F, 8, USER_Q, 0, USR_T, USR_M, 0, "init" },  

06109 };  

06110  

06111 /* Verify the size of the system image table at compile time. Also verify that  

06112 * the first chunk of the ipc mask has enough bits to accommodate the processes  

06113 * in the image.  

06114 * If a problem is detected, the size of the 'dummy' array will be negative,  

06115 * causing a compile time error. Note that no space is actually allocated  

06116 * because 'dummy' is declared extern.  

06117 */  

06118 extern int dummy[(NR_BOOT_PROCS==sizeof(image)/  

06119 sizeof(struct boot_image))?1:-1];  

06120 extern int dummy[(BITCHUNK_BITS > NR_BOOT_PROCS - 1) ? 1 : -1];  

06121

```

```
+++++
kernel/mpx.s
+++++
```

```

06200 #  

06201 ! Chooses between the 8086 and 386 versions of the Minix startup code.  

06202  

06203 #include <minix/config.h>  

06204 #if _WORD_SIZE == 2  

06205 #include "mpx88.s"  

06206 #else  

06207 #include "mpx386.s"  

06208 #endif

```

```
+++++
kernel/mpx386.s
+++++
```

```

06300 #  

06301 ! This file, mpx386.s, is included by mpx.s when Minix is compiled for  

06302 ! 32-bit Intel CPUs. The alternative mpx88.s is compiled for 16-bit CPUs.  

06303  

06304 ! This file is part of the lowest layer of the MINIX kernel. (The other part  

06305 ! is "proc.c".) The lowest layer does process switching and message handling.  

06306 ! Furthermore it contains the assembler startup code for Minix and the 32-bit  

06307 ! interrupt handlers. It cooperates with the code in "start.c" to set up a  

06308 ! good environment for main().  

06309  

06310 ! Every transition to the kernel goes through this file. Transitions to the  

06311 ! kernel may be nested. The initial entry may be with a system call (i.e.,  

06312 ! send or receive a message), an exception or a hardware interrupt; kernel  

06313 ! reentries may only be made by hardware interrupts. The count of reentries  

06314 ! is kept in "k_reenter". It is important for deciding whether to switch to  

06315 ! the kernel stack and for protecting the message passing code in "proc.c".  

06316  

06317 ! For the message passing trap, most of the machine state is saved in the  

06318 ! proc table. (Some of the registers need not be saved.) Then the stack is  

06319 ! switched to "k_stack", and interrupts are reenabled. Finally, the system  

06320 ! call handler (in C) is called. When it returns, interrupts are disabled  

06321 ! again and the code falls into the restart routine, to finish off held-up

```

```

06322 ! interrupts and run the process or task whose pointer is in "proc_ptr".
06323
06324 ! Hardware interrupt handlers do the same, except (1) The entire state must
06325 ! be saved. (2) There are too many handlers to do this inline, so the save
06326 ! routine is called. A few cycles are saved by pushing the address of the
06327 ! appropriate restart routine for a return later. (3) A stack switch is
06328 ! avoided when the stack is already switched. (4) The (master) 8259 interrupt
06329 ! controller is reenabled centrally in save(). (5) Each interrupt handler
06330 ! masks its interrupt line using the 8259 before enabling other unmasked)
06331 ! interrupts, and unmasks it after servicing the interrupt. This limits the
06332 ! nest level to the number of lines and protects the handler from itself.
06333
06334 ! For communication with the boot monitor at startup time some constant
06335 ! data are compiled into the beginning of the text segment. This facilitates
06336 ! reading the data at the start of the boot process, since only the first
06337 ! sector of the file needs to be read.
06338
06339 ! Some data storage is also allocated at the end of this file. This data
06340 ! will be at the start of the data segment of the kernel and will be read
06341 ! and modified by the boot monitor before the kernel starts.
06342
06343 ! sections
06344
06345 .sect .text
06346 begtext:
06347 .sect .rom
06348 begrom:
06349 .sect .data
06350 begdata:
06351 .sect .bss
06352 begbss:
06353
06354 #include <minix/config.h>
06355 #include <minix/const.h>
06356 #include <minix/com.h>
06357 #include <ibm/interrupt.h>
06358 #include "const.h"
06359 #include "protect.h"
06360 #include "sconst.h"
06361
06362 /* Selected 386 tss offsets. */
06363 #define TSS3_S_SPO 4
06364
06365 ! Exported functions
06366 ! Note: in assembly language the .define statement applied to a function name
06367 ! is loosely equivalent to a prototype in C code -- it makes it possible to
06368 ! link to an entity declared in the assembly code but does not create
06369 ! the entity.
06370
06371 .define _restart
06372 .define save
06373
06374 .define _divide_error
06375 .define _single_step_exception
06376 .define _nmi
06377 .define _breakpoint_exception
06378 .define _overflow
06379 .define _bounds_check
06380 .define _invalid_opcode
06381 .define _copr_not_available
06382 .define _double_fault
06383 .define _copr_seg_overrun
06384 .define _invalid_tss
06385 .define _segment_not_present
06386 .define _stack_exception
06387 .define _general_protection
06388 .define _page_fault

```

```

06389 .define _copr_error
06390
06391 .define _hwint00      ! handlers for hardware interrupts
06392 .define _hwint01
06393 .define _hwint02
06394 .define _hwint03
06395 .define _hwint04
06396 .define _hwint05
06397 .define _hwint06
06398 .define _hwint07
06399 .define _hwint08
06400 .define _hwint09
06401 .define _hwint10
06402 .define _hwint11
06403 .define _hwint12
06404 .define _hwint13
06405 .define _hwint14
06406 .define _hwint15
06407
06408 .define _s_call
06409 .define _p_s_call
06410 .define _level0_call
06411
06412 ! Exported variables.
06413 .define begbss
06414 .define begdata
06415
06416 .sect .text
06417 !*=====
06418 !*          MINIX
06419 !*=====
06420 MINIX:           ! this is the entry point for the MINIX kernel
06421     jmp    over_flags   ! skip over the next few bytes
06422     .data2 CLICK_SHIFT ! for the monitor: memory granularity
06423 flags:
06424     .data2 0x01FD       ! boot monitor flags:
06425             !      call in 386 mode, make bss, make stack,
06426             !      load high, don't patch, will return,
06427             !      uses generic INT, memory vector,
06428             !      new boot code return
06429     nop               ! extra byte to sync up disassembler
06430 over_flags:
06431
06432 ! Set up a C stack frame on the monitor stack. (The monitor sets cs and ds
06433 ! right. The ss descriptor still references the monitor data segment.)
06434     movzx  esp, sp       ! monitor stack is a 16 bit stack
06435     push   ebp
06436     mov    ebp, esp
06437     push   esi
06438     push   edi
06439     cmp    4(ebp), 0     ! monitor return vector is
06440     jz     noret        ! nonzero if return possible
06441     inc    (_mon_return)
06442 noret:  mov    (_mon_sp), esp ! save stack pointer for later return
06443
06444 ! Copy the monitor global descriptor table to the address space of kernel and
06445 ! switch over to it. Prot_init() can then update it with immediate effect.
06446
06447     sgdt   (_gdt+GDT_SELECTOR)      ! get the monitor gdtr
06448     mov    esi, (_gdt+GDT_SELECTOR+2) ! absolute address of GDT
06449     mov    ebx, _gdt                ! address of kernel GDT
06450     mov    ecx, 8*8                 ! copying eight descriptors
06451 copygdt:
06452     eseg   movb  al, (esi)
06453             movb  (ebx), al
06454             inc   esi
06455             inc   ebx

```

```

06456    loop    copygdt
06457    mov     eax, (_gdt+DS_SELECTOR+2)      ! base of kernel data
06458    and     eax, 0x00FFFFFF                 ! only 24 bits
06459    add     eax, _gdt                      ! eax = vir2phys(gdt)
06460    mov     (_gdt+GDT_SELECTOR+2), eax       ! set base of GDT
06461    lgdt   (_gdt+GDT_SELECTOR)             ! switch over to kernel GDT
06462
06463 ! Locate boot parameters, set up kernel segment registers and stack.
06464    mov     ebx, 8(ebp)          ! boot parameters offset
06465    mov     edx, 12(ebp)         ! boot parameters length
06466    mov     eax, 16(ebp)         ! address of a.out headers
06467    mov     (_aout), eax
06468    mov     ax, ds              ! kernel data
06469    mov     es, ax
06470    mov     fs, ax
06471    mov     gs, ax
06472    mov     ss, ax
06473    mov     esp, k_stktop      ! set sp to point to the top of kernel stack
06474
06475 ! Call C startup code to set up a proper environment to run main().
06476    push   edx
06477    push   ebx
06478    push   SS_SELECTOR
06479    push   DS_SELECTOR
06480    push   CS_SELECTOR
06481    call   _cstart           ! cstart(cs, ds, mds, parmoff, parmlen)
06482    add    esp, 5*4
06483
06484 ! Reload gdtr, idtr and the segment registers to global descriptor table set
06485 ! up by prot_init().
06486
06487    lgdt   (_gdt+GDT_SELECTOR)
06488    lidt   (_gdt+IDT_SELECTOR)
06489
06490    jmpf   CS_SELECTOR:csinit
06491 csinit:
06492    o16   mov     ax, DS_SELECTOR
06493    mov     ds, ax
06494    mov     es, ax
06495    mov     fs, ax
06496    mov     gs, ax
06497    mov     ss, ax
06498    o16   mov     ax, TSS_SELECTOR        ! no other TSS is used
06499    ltr    ax
06500    push   0                         ! set flags to known good state
06501    popf
06502
06503    jmp   _main                     ! main()
06504
06505
06506 !=====
06507 !*          interrupt handlers
06508 !*          interrupt handlers for 386 32-bit protected mode
06509 !*=====
06510
06511 !=====
06512 !*          hwint00 - 07
06513 !=====
06514 ! Note this is a macro, it just looks like a subroutine.
06515 #define hwint_master(irq)    \
06516    call   save               /* save interrupted process state */; \
06517    push   (_irq_handlers+4*irq) /* irq_handlers[irq] */; \
06518    call   _intr_handle        /* intr_handle(irq_handlers[irq]) */; \
06519    pop    ecx
06520    cmp    (_irq_actids+4*irq), 0 /* interrupt still active? */; \
06521    jz    of                  ; \
06522    inb   INT_CTMASK         /* get current mask */; \

```

```

06523     orb    al, [1<<irq]      /* mask irq */          ;\
06524     outb   INT_CTLMASK        /* disable the irq */    *;/\
06525 0:    movb   al, END_OF_INT   ;\
06526     outb   INT_CTL           /* reenable master 8259 */ *;/\
06527     ret                /* restart (another) process */ *;/\
06528
06529 ! Each of these entry points is an expansion of the hwint_master macro
06530     .align 16
06531 _hwint00:           ! Interrupt routine for irq 0 (the clock).
06532     hwint_master(0)
06533
06534     .align 16
06535 _hwint01:           ! Interrupt routine for irq 1 (keyboard)
06536     hwint_master(1)
06537
06538     .align 16
06539 _hwint02:           ! Interrupt routine for irq 2 (cascade!)
06540     hwint_master(2)
06541
06542     .align 16
06543 _hwint03:           ! Interrupt routine for irq 3 (second serial)
06544     hwint_master(3)
06545
06546     .align 16
06547 _hwint04:           ! Interrupt routine for irq 4 (first serial)
06548     hwint_master(4)
06549
06550     .align 16
06551 _hwint05:           ! Interrupt routine for irq 5 (XT winchester)
06552     hwint_master(5)
06553
06554     .align 16
06555 _hwint06:           ! Interrupt routine for irq 6 (floppy)
06556     hwint_master(6)
06557
06558     .align 16
06559 _hwint07:           ! Interrupt routine for irq 7 (printer)
06560     hwint_master(7)
06561
06562 !*=====
06563 !*          hwint08 - 15
06564 !*=====
06565 ! Note this is a macro, it just looks like a subroutine.
06566 #define hwint_slave(irq) \
06567     call  save             /* save interrupted process state */;\ \
06568     push  (_irq_handlers+4*irq) /* irq_handlers[irq] */;\ \
06569     call  _intr_handle      /* intr_handle(irq_handlers[irq]) */;\ \
06570     pop   ecx              ;\
06571     cmp   (_irq_actids+4*irq), 0 /* interrupt still active? */;\ \
06572     jz    Of               ;\
06573     inb   INT2_CTLMASK     ;\
06574     orb   al, [1<<[irq-8]] ;\
06575     outb  INT2_CTLMASK     /* disable the irq */;\ \
06576 0:    movb   al, END_OF_INT   ;\
06577     outb  INT_CTL           /* reenable master 8259 */;\ \
06578     outb  INT2_CTL           /* reenable slave 8259 */;\ \
06579     ret                /* restart (another) process */;\ \
06580
06581 ! Each of these entry points is an expansion of the hwint_slave macro
06582     .align 16
06583 _hwint08:           ! Interrupt routine for irq 8 (realtime clock)
06584     hwint_slave(8)
06585
06586     .align 16
06587 _hwint09:           ! Interrupt routine for irq 9 (irq 2 redirected)
06588     hwint_slave(9)
06589

```

```

06590      .align 16
06591 _hwint10:           ! Interrupt routine for irq 10
06592     hwind_slave(10)
06593
06594      .align 16
06595 _hwint11:           ! Interrupt routine for irq 11
06596     hwind_slave(11)
06597
06598      .align 16
06599 _hwint12:           ! Interrupt routine for irq 12
06600     hwind_slave(12)
06601
06602      .align 16
06603 _hwint13:           ! Interrupt routine for irq 13 (FPU exception)
06604     hwind_slave(13)
06605
06606      .align 16
06607 _hwint14:           ! Interrupt routine for irq 14 (AT winchester)
06608     hwind_slave(14)
06609
06610      .align 16
06611 _hwint15:           ! Interrupt routine for irq 15
06612     hwind_slave(15)
06613
06614 !*=====
06615 !*          save
06616 !*=====
06617 ! Save for protected mode.
06618 ! This is much simpler than for 8086 mode, because the stack already points
06619 ! into the process table, or has already been switched to the kernel stack.
06620
06621      .align 16
06622 save:
06623     cld                      ! set direction flag to a known value
06624     pushad                   ! save "general" registers
06625     o16 push    ds            ! save ds
06626     o16 push    es            ! save es
06627     o16 push    fs            ! save fs
06628     o16 push    gs            ! save gs
06629     mov    dx, ss             ! ss is kernel data segment
06630     mov    ds, dx             ! load rest of kernel segments
06631     mov    es, dx             ! kernel does not use fs, gs
06632     mov    eax, esp            ! prepare to return
06633     incb   (_k_reenter)       ! from -1 if not reentering
06634     jnz    set_restart1        ! stack is already kernel stack
06635     mov    esp, k_stktop
06636     push   _restart          ! build return address for int handler
06637     xor    ebp, ebp            ! for stacktrace
06638     jmp    RETADR-P_STACKBASE(eax)
06639
06640      .align 4
06641 set_restart1:
06642     push   restart1
06643     jmp    RETADR-P_STACKBASE(eax)
06644
06645 !*=====
06646 !*          _s_call
06647 !*=====
06648      .align 16
06649 _s_call:
06650 _p_s_call:
06651     cld                      ! set direction flag to a known value
06652     sub    esp, 6*4            ! skip RETADR, eax, ecx, edx, ebx, est
06653     push   ebp
06654     push   esi
06655     push   edi
06656     o16 push   ds
06657     o16 push   es

```

```

06658    o16 push    fs
06659    o16 push    gs
06660        mov     dx, ss
06661        mov     ds, dx
06662        mov     es, dx
06663        incb   (_k_reenter)
06664        mov     esi, esp      ! assumes P_STACKBASE == 0
06665        mov     esp, k_stktop
06666        xor     ebp, ebp      ! for stacktrace
06667                ! end of inline save
06668                ! now set up parameters for sys_call()
06669        push    ebx      ! pointer to user message
06670        push    eax      ! src/dest
06671        push    ecx      ! SEND/RECEIVE/BOTH
06672        call    _sys_call      ! sys_call(function, src_dest, m_ptr)
06673                ! caller is now explicitly in proc_ptr
06674        mov     AXREG(es1), eax ! sys_call MUST PRESERVE si
06675
06676 ! Fall into code to restart proc/task running.
06677
06678 !***** restart *****
06679 !*                         restart
06680 !*****
06681 _restart:
06682
06683 ! Restart the current process or the next process if it is set.
06684
06685        cmp     (_next_ptr), 0      ! see if another process is scheduled
06686        jz      Of
06687        mov     eax, (_next_ptr)
06688        mov     (_proc_ptr), eax      ! schedule new process
06689        mov     (_next_ptr), 0
06690 O:       mov     esp, (_proc_ptr)      ! will assume P_STACKBASE == 0
06691        lldt   P_LDT_SEL(esp)      ! enable process' segment descriptors
06692        lea     eax, P_STACKTOP(esp)  ! arrange for next interrupt
06693        mov     (_tss+TSS3_S_SP0), eax ! to save state in process table
06694 _restart1:
06695        decb   (_k_reenter)
06696        o16 pop    gs
06697        o16 pop    fs
06698        o16 pop    es
06699        o16 pop    ds
06700        popad
06701        add    esp, 4      ! skip return adr
06702        iretd
06703                ! continue process
06704 !***** exception handlers *****
06705 !*                         exception handlers
06706 !*****
06707 _divide_error:
06708        push    DIVIDE_VECTOR
06709        jmp     exception
06710
06711 _single_step_exception:
06712        push    DEBUG_VECTOR
06713        jmp     exception
06714
06715 _nmi:
06716        push    NMI_VECTOR
06717        jmp     exception
06718
06719 _breakpoint_exception:
06720        push    BREAKPOINT_VECTOR
06721        jmp     exception
06722
06723 _overflow:
06724        push    OVERFLOW_VECTOR
06725        jmp     exception

```

```

06726
06727     _bounds_check:
06728         push    BOUNDS_VECTOR
06729         jmp     exception
06730
06731     _invalid_opcode:
06732         push    INVAL_OP_VECTOR
06733         jmp     exception
06734
06735     _coproc_not_available:
06736         push    COPROC_NOT_VECTOR
06737         jmp     exception
06738
06739     _double_fault:
06740         push    DOUBLEFAULT_VECTOR
06741         jmp     errexception
06742
06743     _coproc_seg_overrun:
06744         push    COPROC_SEG_VECTOR
06745         jmp     exception
06746
06747     _invalid_tss:
06748         push    INVAL_TSS_VECTOR
06749         jmp     errexception
06750
06751     _segment_not_present:
06752         push    SEG_NOT_VECTOR
06753         jmp     errexception
06754
06755     _stack_exception:
06756         push    STACK_FAULT_VECTOR
06757         jmp     errexception
06758
06759     _general_protection:
06760         push    PROTECTION_VECTOR
06761         jmp     errexception
06762
06763     _page_fault:
06764         push    PAGE_FAULT_VECTOR
06765         jmp     errexception
06766
06767     _coproc_error:
06768         push    COPROC_ERR_VECTOR
06769         jmp     exception
06770
06771 !*=====
06772 !*           exception
06773 !*=====
06774 ! This is called for all exceptions which do not push an error code.
06775
06776     .align 16
06777 exception:
06778     sseg   mov     (trap_errno), 0          ! clear trap_errno
06779     sseg   pop     (ex_number)
06780     jmp    exception1
06781
06782 !*=====
06783 !*           errexception
06784 !*=====
06785 ! This is called for all exceptions which push an error code.
06786
06787     .align 16
06788 errexception:
06789     sseg   pop     (ex_number)
06790     sseg   pop     (trap_errno)
06791 exception1:                           ! Common for all exceptions.
06792     push    eax                         ! eax is scratch register

```

```

06793     mov    eax, 0+4(esp)          ! old eip
06794 sseg   mov    (old_eip), eax
06795     movzx eax, 4+4(esp)         ! old cs
06796 sseg   mov    (old_cs), eax
06797     mov    eax, 8+4(esp)         ! old eflags
06798 sseg   mov    (old_eflags), eax
06799     pop    eax
06800     call   save
06801     push   (old_eflags)
06802     push   (old_cs)
06803     push   (old_eip)
06804     push   (trap_errno)
06805     push   (ex_number)
06806     call   _exception          ! (ex_number, trap_errno, old_eip,
06807                                     !           old_cs, old_eflags)
06808     add    esp, 5*4
06809     ret
06810

06811 !*****+
06812 !*          level0_call          *
06813 !*****+
06814 _level0_call:
06815     call   save
06816     jmp    (_level0_func)
06817

06818 !*****+
06819 !*          data               *
06820 !*****+
06821
06822 .sect .rom      ! Before the string table please
06823     .data2 0x526F            ! this must be the first data entry (magic #)
06824
06825 .sect .bss
06826 k_stack:
06827     .space K_STACK_BYTES      ! kernel stack
06828 k_stktop:
06829     .comm   ex_number, 4
06830     .comm   trap_errno, 4
06831     .comm   old_eip, 4
06832     .comm   old_cs, 4
06833     .comm   old_eflags, 4

+++++++
kernel/start.c
+++++++
06900 /* This file contains the C startup code for Minix on Intel processors.
06901 * It cooperates with mpx.s to set up a good environment for main().
06902 *
06903 * This code runs in real mode for a 16 bit kernel and may have to switch
06904 * to protected mode for a 286.
06905 * For a 32 bit kernel this already runs in protected mode, but the selectors
06906 * are still those given by the BIOS with interrupts disabled, so the
06907 * descriptors need to be reloaded and interrupt descriptors made.
06908 */
06909
06910 #include "kernel.h"
06911 #include "protect.h"
06912 #include "proc.h"
06913 #include <stdlib.h>
06914 #include <string.h>
06915
06916 FORWARD _PROTOTYPE( char *get_value, (_CONST char *params, _CONST char *key));
06917 /*=====
06918 *          cstart
06919 *=====

```

```

06920 PUBLIC void cstart(cs, ds, mds, parmoff, parmsize)
06921 U16_t cs, ds;           /* kernel code and data segment */
06922 U16_t mds;             /* monitor data segment */
06923 U16_t parmoff, parmsize; /* boot parameters offset and length */
06924 {
06925 /* Perform system initializations prior to calling main(). Most settings are
06926 * determined with help of the environment strings passed by MINIX' loader.
06927 */
06928 char params[128*sizeof(char *)];           /* boot monitor parameters */
06929 register char *value;                      /* value in key=value pair */
06930 extern int etext, end;
06931
06932 /* Decide if mode is protected; 386 or higher implies protected mode.
06933 * This must be done first, because it is needed for, e.g., seg2phys().
06934 * For 286 machines we cannot decide on protected mode, yet. This is
06935 * done below.
06936 */
06937 #if _WORD_SIZE != 2
06938     machine.protected = 1;
06939 #endif
06940
06941 /* Record where the kernel and the monitor are. */
06942 kinfo.code_base = seg2phys(cs);
06943 kinfo.code_size = (phys_bytes) &etext;          /* size of code segment */
06944 kinfo.data_base = seg2phys(ds);
06945 kinfo.data_size = (phys_bytes) &end;           /* size of data segment */
06946
06947 /* Initialize protected mode descriptors. */
06948 prot_init();
06949
06950 /* Copy the boot parameters to the local buffer. */
06951 kinfo.params_base = seg2phys(mds) + parmoff;
06952 kinfo.params_size = MIN(parmsize,sizeof(params)-2);
06953 phys_copy(kinfo.params_base, vir2phys(params), kinfo.params_size);
06954
06955 /* Record miscellaneous information for user-space servers. */
06956 kinfo.nr_procs = NR_PROCS;
06957 kinfo.nr_tasks = NR_TASKS;
06958 strncpy(kinfo.release, OS_RELEASE, sizeof(kinfo.release));
06959 kinfo.release[sizeof(kinfo.release)-1] = '\0';
06960 strncpy(kinfo.version, OS_VERSION, sizeof(kinfo.version));
06961 kinfo.version[sizeof(kinfo.version)-1] = '\0';
06962 kinfo.proc_addr = (vir_bytes) proc;
06963 kinfo.kmem_base = vir2phys(0);
06964 kinfo.kmem_size = (phys_bytes) &end;
06965
06966 /* Processor? 86, 186, 286, 386, ...
06967 * Decide if mode is protected for older machines.
06968 */
06969 machine.processor=atoi(get_value(params, "processor"));
06970 #if _WORD_SIZE == 2
06971     machine.protected = machine.processor >= 286;
06972 #endif
06973     if (! machine.protected) mon_return = 0;
06974
06975 /* XT, AT or MCA bus? */
06976 value = get_value(params, "bus");
06977 if (value == NIL_PTR || strcmp(value, "at") == 0) {
06978     machine.pc_at = TRUE;                  /* PC-AT compatible hardware */
06979 } else if (strcmp(value, "mca") == 0) {
06980     machine.pc_at = machine.ps_mca = TRUE; /* PS/2 with micro channel */
06981 }
06982
06983 /* Type of VDU: */
06984 value = get_value(params, "video");           /* EGA or VGA video unit */
06985 if (strcmp(value, "ega") == 0) machine.vdu_ega = TRUE;
06986 if (strcmp(value, "vga") == 0) machine.vdu_vga = machine.vdu_ega = TRUE;
06987

```

```

06988 /* Return to assembler code to switch to protected mode (if 286),
06989 * reload selectors and call main().
06990 */
06991 }

06993 /*=====
06994 *           get_value
06995 =====*/
06996
06997 PRIVATE char *get_value(params, name)
06998 _CONST char *params;                      /* boot monitor parameters */
06999 _CONST char *name;                        /* key to look up */
07000 {
07001 /* Get environment value - kernel version of getenv to avoid setting up the
07002 * usual environment array.
07003 */
07004 register _CONST char *namep;
07005 register char *envp;
07006
07007 for (envp = (char *) params; *envp != 0;) {
07008     for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
07009         ;
07010     if (*namep == '\0' && *envp == '=') return(envp + 1);
07011     while (*envp++ != 0)
07012         ;
07013 }
07014 return(NIL_PTR);
07015 }

```

=====  
kernel/main.c  
=====

```

07100 /* This file contains the main program of MINIX as well as its shutdown code.
07101 * The routine main() initializes the system and starts the ball rolling by
07102 * setting up the process table, interrupt vectors, and scheduling each task
07103 * to run to initialize itself.
07104 * The routine shutdown() does the opposite and brings down MINIX.
07105 *
07106 * The entries into this file are:
07107 *   main:          MINIX main program
07108 *   prepare_shutdown:  prepare to take MINIX down
07109 *
07110 * Changes:
07111 *   Nov 24, 2004  simplified main() with system image (Jorrit N. Herder)
07112 *   Aug 20, 2004  new prepare_shutdown() and shutdown() (Jorrit N. Herder)
07113 */
07114 #include "kernel.h"
07115 #include <signal.h>
07116 #include <string.h>
07117 #include <unistd.h>
07118 #include <a.out.h>
07119 #include <minix/callnr.h>
07120 #include <minix/com.h>
07121 #include "proc.h"
07122
07123 /* Prototype declarations for PRIVATE functions. */
07124 FORWARD _PROTOTYPE( void announce, (void));
07125 FORWARD _PROTOTYPE( void shutdown, (timer_t *tp));
07126
07127 /*=====
07128 *           main
07129 =====*/

```

```

07130 PUBLIC void main()
07131 {
07132 /* Start the ball rolling. */
07133 struct boot_image *ip;           /* boot image pointer */
07134 register struct proc *rp;       /* process pointer */
07135 register struct priv *sp;       /* privilege structure pointer */
07136 register int i, s;
07137 int hdrindex;                 /* index to array of a.out headers */
07138 phys_ticks text_base;
07139 vir_ticks text_ticks, data_ticks;
07140 reg_t ktsb;                   /* kernel task stack base */
07141 struct exec e_hdr;            /* for a copy of an a.out header */
07142
07143 /* Initialize the interrupt controller. */
07144 intr_init(1);
07145
07146 /* Clear the process table. Anounce each slot as empty and set up mappings
07147 * for proc_addr() and proc_nr() macros. Do the same for the table with
07148 * privilege structures for the system processes.
07149 */
07150 for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
07151     rp->p_rts_flags = SLOT_FREE;          /* initialize free slot */
07152     rp->p_nr = i;                      /* proc number from ptr */
07153     (pproc_addr + NR_TASKS)[i] = rp;      /* proc ptr from number */
07154 }
07155 for (sp = BEG_PRIV_ADDR, i = 0; sp < END_PRIV_ADDR; ++sp, ++i) {
07156     sp->s_proc_nr = NONE;                /* initialize as free */
07157     sp->s_id = i;                      /* priv structure index */
07158     ppriv_addr[i] = sp;                  /* priv ptr from number */
07159 }
07160
07161 /* Set up proc table entries for tasks and servers. The stacks of the
07162 * kernel tasks are initialized to an array in data space. The stacks
07163 * of the servers have been added to the data segment by the monitor, so
07164 * the stack pointer is set to the end of the data segment. All the
07165 * processes are in low memory on the 8086. On the 386 only the kernel
07166 * is in low memory, the rest is loaded in extended memory.
07167 */
07168
07169 /* Task stacks. */
07170 ktsb = (reg_t) t_stack;
07171
07172 for (i=0; i < NR_BOOT_PROCS; ++i) {
07173     ip = &image[i];                     /* process' attributes */
07174     rp = proc_addr(ip->proc_nr);        /* get process pointer */
07175     rp->p_max_priority = ip->priority;  /* max scheduling priority */
07176     rp->p_priority = ip->priority;      /* current priority */
07177     rp->p_quantum_size = ip->quantum;   /* quantum size in ticks */
07178     rp->p_ticks_left = ip->quantum;     /* current credit */
07179     strncpy(rp->p_name, ip->proc_name, P_NAME_LEN); /* set process name */
07180     (void) get_priv(rp, (ip->flags & SYS_PROC)); /* assign structure */
07181     priv(rp)->s_flags = ip->flags;        /* process flags */
07182     priv(rp)->s_trap_mask = ip->trap_mask; /* allowed traps */
07183     priv(rp)->s_call_mask = ip->call_mask; /* kernel call mask */
07184     priv(rp)->s_ipc_to.chunk[0] = ip->ipc_to; /* restrict targets */
07185     if (iskerneln(proc_nr(rp))) {          /* part of the kernel? */
07186         if (ip->stksize > 0) {             /* HARDWARE stack size is 0 */
07187             rp->p_priv->s_stack_guard = (reg_t *) ktsb;
07188             *rp->p_priv->s_stack_guard = STACK_GUARD;
07189         }
07190         ktsb += ip->stksize;    /* point to high end of stack */
07191         rp->p_reg.sp = ktsb;    /* this task's initial stack ptr */
07192         text_base = kinfo.code_base >> CLICK_SHIFT;
07193                                         /* processes that are in the kernel */
07194         hdrindex = 0;           /* all use the first a.out header */
07195     } else {
07196         hdrindex = 1 + i-NR_TASKS; /* servers, drivers, INIT */

```

```

07197         }
07198
07199     /* The bootstrap loader created an array of the a.out headers at
07200     * absolute address 'aout'. Get one element to e_hdr.
07201     */
07202     phys_copy(aout + hdrindex * A_MINHDR, vir2phys(&e_hdr),
07203             (phys_bytes) A_MINHDR);
07204     /* Convert addresses to clicks and build process memory map */
07205     text_base = e_hdr.a_syms >> CLICK_SHIFT;
07206     text_clicks = (e_hdr.a_text + CLICK_SIZE-1) >> CLICK_SHIFT;
07207     if (!(e_hdr.a_flags & A_SEP)) text_clicks = 0; /* common I&D */
07208     data_clicks = (e_hdr.a_total + CLICK_SIZE-1) >> CLICK_SHIFT;
07209     rp->p_memmap[T].mem_phys = text_base;
07210     rp->p_memmap[T].mem_len = text_clicks;
07211     rp->p_memmap[D].mem_phys = text_base + text_clicks;
07212     rp->p_memmap[D].mem_len = data_clicks;
07213     rp->p_memmap[S].mem_phys = text_base + text_clicks + data_clicks;
07214     rp->p_memmap[S].mem_vir = data_clicks; /* empty - stack is in data */
07215
07216     /* Set initial register values. The processor status word for tasks
07217     * is different from that of other processes because tasks can
07218     * access I/O; this is not allowed to less-privileged processes
07219     */
07220     rp->p_reg.pc = (reg_t) ip->initial_pc;
07221     rp->p_reg.psw = (iskernelp(rp)) ? INIT_TASK_PSW : INIT_PSW;
07222
07223     /* Initialize the server stack pointer. Take it down one word
07224     * to give crtso.s something to use as "argc".
07225     */
07226     if (isusern(proc_nr(rp))) { /* user-space process? */
07227         rp->p_reg.sp = (rp->p_memmap[S].mem_vir +
07228                         rp->p_memmap[S].mem_len) << CLICK_SHIFT;
07229         rp->p_reg.sp -= sizeof(reg_t);
07230     }
07231
07232     /* Set ready. The HARDWARE task is never ready. */
07233     if (rp->p_nr != HARDWARE) {
07234         rp->p_rts_flags = 0; /* runnable if no flags */
07235         lock_enqueue(rp); /* add to scheduling queues */
07236     } else {
07237         rp->p_rts_flags = NO_MAP; /* prevent from running */
07238     }
07239
07240     /* Code and data segments must be allocated in protected mode. */
07241     alloc_segments(rp);
07242 }
07243
07244 /* We're definitely not shutting down. */
07245 shutdown_started = 0;
07246
07247 /* MINIX is now ready. All boot image processes are on the ready queue.
07248 * Return to the assembly code to start running the current process.
07249 */
07250 bill_ptr = proc_addr(IDLE); /* it has to point somewhere */
07251 announce(); /* print MINIX startup banner */
07252 restart();
07253 }

07255 /*=====
07256 *          announce
07257 *=====
07258 PRIVATE void announce(void)
07259 {
07260     /* Display the MINIX startup banner. */
07261     kprintf("MINIX %s.%s."
07262             "Copyright 2006, Vrije Universiteit, Amsterdam, The Netherlands\n",
07263             OS_RELEASE, OS_VERSION);
07264

```

```

07265     /* Real mode, or 16/32-bit protected mode? */
07266     kprintf("Executing in %s mode.\n\n",
07267         machine.protected ? "32-bit protected" : "real");
07268 }

07270 /*=====
07271 *          prepare_shutdown
07272 *=====
07273 PUBLIC void prepare_shutdown(how)
07274 int how;
07275 {
07276 /* This function prepares to shutdown MINIX. */
07277     static timer_t shutdown_timer;
07278     register struct proc *rp;
07279     message m;

07280 /* Show debugging dumps on panics. Make sure that the TTY task is still
07281 * available to handle them. This is done with help of a non-blocking send.
07282 * We rely on TTY to call sys_abort() when it is done with the dumps.
07283 */
07284 if (how == RBT_PANIC) {
07285     m.m_type = PANIC_DUMPS;
07286     if (nb_send(TTY_PROC_NR,&m)==OK) /* don't block if TTY isn't ready */
07287         return;                      /* await sys_abort() from TTY */
07288 }
07289

07290 /* Send a signal to all system processes that are still alive to inform
07291 * them that the MINIX kernel is shutting down. A proper shutdown sequence
07292 * should be implemented by a user-space server. This mechanism is useful
07293 * as a backup in case of system panics, so that system processes can still
07294 * run their shutdown code, e.g., to synchronize the FS or to let the TTY
07295 * switch to the first console.
07296 */
07297 kprintf("Sending SIGKSTOP to system processes ... \n");
07298 for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
07299     if (!isemptyp(rp) && (priv(rp)->s_flags & SYS_PROC) && !iskernelp(rp))
07300         send_sig(proc_nr(rp), SIGKSTOP);
07301 }
07302

07303 /* We're shutting down. Diagnostics may behave differently now. */
07304 shutdown_started = 1;
07305

07306 /* Notify system processes of the upcoming shutdown and allow them to be
07307 * scheduled by setting a watchdog timer that calls shutdown(). The timer
07308 * argument passes the shutdown status.
07309 */
07310 kprintf("MINIX will now be shut down ... \n");
07311 tmr_arg(&shutdown_timer)->ta_int = how;
07312

07313 /* Continue after 1 second, to give processes a chance to get
07314 * scheduled to do shutdown work.
07315 */
07316 set_timer(&shutdown_timer, get_uptime() + HZ, shutdown);
07317 }

07318 */

07319 /*=====
07320 *          shutdown
07321 *=====
07322 PRIVATE void shutdown(tp)
07323 timer_t *tp;
07324 {
07325 /* This function is called from prepare_shutdown or stop_sequence to bring
07326 * down MINIX. How to shutdown is in the argument: RBT_HALT (return to the
07327 * monitor), RBT_MONITOR (execute given code), RBT_RESET (hard reset).
07328 */
07329     int how = tmr_arg(tp)->ta_int;
07330     u16_t magic;
07331 }
```

```

07332
07333 /* Now mask all interrupts, including the clock, and stop the clock. */
07334 outb(INT_CTLMASK, ~0);
07335 clock_stop();
07336
07337 if (mon_return && how != RBT_RESET) {
07338     /* Reinitialize the interrupt controllers to the BIOS defaults. */
07339     intr_init(0);
07340     outb(INT_CTLMASK, 0);
07341     outb(INT2_CTLMASK, 0);
07342
07343     /* Return to the boot monitor. Set the program if not already done. */
07344     if (how != RBT_MONITOR) phys_copy(vir2phys(""), kinfo.params_base, 1);
07345     level0(monitor);
07346 }
07347
07348 /* Reset the system by jumping to the reset address (real mode), or by
07349 * forcing a processor shutdown (protected mode). First stop the BIOS
07350 * memory test by setting a soft reset flag.
07351 */
07352 magic = STOP_MEM_CHECK;
07353 phys_copy(vir2phys(&magic), SOFT_RESET_FLAG_ADDR, SOFT_RESET_FLAG_SIZE);
07354 level0(reset);
07355 }
```

```
=====
kernel/proc.c
=====

07400 /* This file contains essentially all of the process and message handling.
07401 * Together with "mpx.s" it forms the lowest layer of the MINIX kernel.
07402 * There is one entry point from the outside:
07403 *
07404 *   sys_call:      a system call, i.e., the kernel is trapped with an INT
07405 *
07406 * As well as several entry points used from the interrupt and task level:
07407 *
07408 *   lock_notify:    notify a process of a system event
07409 *   lock_send:      send a message to a process
07410 *   lock_enqueue:   put a process on one of the scheduling queues
07411 *   lock_dequeue:   remove a process from the scheduling queues
07412 *
07413 * Changes:
07414 *   Aug 19, 2005   rewrote scheduling code (Jorrit N. Herder)
07415 *   Jul 25, 2005   rewrote system call handling (Jorrit N. Herder)
07416 *   May 26, 2005   rewrote message passing functions (Jorrit N. Herder)
07417 *   May 24, 2005   new notification system call (Jorrit N. Herder)
07418 *   Oct 28, 2004   nonblocking send and receive calls (Jorrit N. Herder)
07419 *
07420 * The code here is critical to make everything work and is important for the
07421 * overall performance of the system. A large fraction of the code deals with
07422 * list manipulation. To make this both easy to understand and fast to execute
07423 * pointer pointers are used throughout the code. Pointer pointers prevent
07424 * exceptions for the head or tail of a linked list.
07425 *
07426 * node_t *queue, *new_node; // assume these as global variables
07427 * node_t **xpp = &queue; // get pointer pointer to head of queue
07428 * while (*xpp != NULL) // find last pointer of the linked list
07429 *     xpp = &(*xpp)->next; // get pointer to next pointer
07430 *     *xpp = new_node; // now replace the end (the NULL pointer)
07431 *     new_node->next = NULL; // and mark the new end of the list
07432 *
07433 * For example, when adding a new node to the end of the list, one normally
```

```

07434 * makes an exception for an empty list and looks up the end of the list for
07435 * nonempty lists. As shown above, this is not required with pointer pointers.
07436 */
07437
07438 #include <minix/com.h>
07439 #include <minix/callnr.h>
07440 #include "kernel.h"
07441 #include "proc.h"
07442
07443 /* Scheduling and message passing functions. The functions are available to
07444 * other parts of the kernel through lock_...(). The lock temporarily disables
07445 * interrupts to prevent race conditions.
07446 */
07447 FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dst,
07448                      message *m_ptr, unsigned flags) );
07449 FORWARD _PROTOTYPE( int mini_receive, (struct proc *caller_ptr, int src,
07450                      message *m_ptr, unsigned flags) );
07451 FORWARD _PROTOTYPE( int mini_notify, (struct proc *caller_ptr, int dst) );
07452
07453 FORWARD _PROTOTYPE( void enqueue, (struct proc *rp) );
07454 FORWARD _PROTOTYPE( void dequeue, (struct proc *rp) );
07455 FORWARD _PROTOTYPE( void sched, (struct proc *rp, int *queue, int *front) );
07456 FORWARD _PROTOTYPE( void pick_proc, (void) );
07457
07458 #define BuildMess(m_ptr, src, dst_ptr) \
07459     (m_ptr)->m_source = (src); \
07460     (m_ptr)->m_type = NOTIFY_FROM(src); \
07461     (m_ptr)->NOTIFY_TIMESTAMP = get_uptime(); \
07462     switch (src) { \
07463         case HARDWARE: \
07464             (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_int_pending; \
07465             priv(dst_ptr)->s_int_pending = 0; \
07466             break; \
07467         case SYSTEM: \
07468             (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_sig_pending; \
07469             priv(dst_ptr)->s_sig_pending = 0; \
07470             break; \
07471     } \
07472
07473 #define CopyMess(s,sp,sm,dp,dm) \
07474     cp_mess(s, (sp)->p_memmap[D].mem_phys, \
07475             (vir_bytes)sm, (dp)->p_memmap[D].mem_phys, (vir_bytes)dm)
07476
07477 /*=====
07478 *                               sys_call
07479 =====*/
07480 PUBLIC int sys_call(call_nr, src_dst, m_ptr)
07481 int call_nr;           /* system call number and flags */
07482 int src_dst;          /* src to receive from or dst to send to */
07483 message *m_ptr;       /* pointer to message in the caller's space */
07484 {
07485 /* System calls are done by trapping to the kernel with an INT instruction.
07486 * The trap is caught and sys_call() is called to send or receive a message
07487 * (or both). The caller is always given by 'proc_ptr'.
07488 */
07489 register struct proc *caller_ptr = proc_ptr; /* get pointer to caller */
07490 int function = call_nr & SYSCALL_FUNC;        /* get system call function */
07491 unsigned flags = call_nr & SYSCALL_FLAGS;      /* get flags */
07492 int mask_entry;                            /* bit to check in send mask */
07493 int result;                                /* the system call's result */
07494 vir_ticks vlo, vhi;                         /* virtual ticks containing message to send */
07495
07496 /* Check if the process has privileges for the requested call. Calls to the
07497 * kernel may only be SENDREC, because tasks always reply and may not block
07498 * if the caller doesn't do receive().
07499 */
07500 if (! (priv(caller_ptr)->s_trap_mask & (1 << function)) ||
07501     (iskerneln(src_dst) && function != SENDREC)

```

```

07502     && function != RECEIVE)) {
07503         kprintf("sys_call: trap %d not allowed, caller %d, src_dst %d\n",
07504             function, proc_nr(caller_ptr), src_dst);
07505         return(ECALLDENIED);           /* trap denied by mask or kernel */
07506     }
07507
07508     /* Require a valid source and/ or destination process, unless echoing. */
07509     if (! (isokprocn(src_dst) || src_dst == ANY || function == ECHO)) {
07510         kprintf("sys_call: invalid src_dst, src_dst %d, caller %d\n",
07511             src_dst, proc_nr(caller_ptr));
07512         return(EBADSRCDST);          /* invalid process number */
07513     }
07514
07515     /* If the call involves a message buffer, i.e., for SEND, RECEIVE, SENDREC,
07516     * or ECHO, check the message pointer. This check allows a message to be
07517     * anywhere in data or stack or gap. It will have to be made more elaborate
07518     * for machines which don't have the gap mapped.
07519     */
07520     if (function & CHECK_PTR) {
07521         vlo = (vir_bytes) m_ptr >> CLICK_SHIFT;
07522         vhi = ((vir_bytes) m_ptr + MESS_SIZE - 1) >> CLICK_SHIFT;
07523         if (vlo < caller_ptr->p_memmap[D].mem_vir || vlo > vhi ||
07524             vhi > caller_ptr->p_memmap[S].mem_vir +
07525             caller_ptr->p_memmap[S].mem_len) {
07526             kprintf("sys_call: invalid message pointer, trap %d, caller %d\n",
07527                 function, proc_nr(caller_ptr));
07528             return(EFAULT);            /* invalid message pointer */
07529         }
07530     }
07531
07532     /* If the call is to send to a process, i.e., for SEND, SENDREC or NOTIFY,
07533     * verify that the caller is allowed to send to the given destination and
07534     * that the destination is still alive.
07535     */
07536     if (function & CHECK_DST) {
07537         if (! get_sys_bit(priv(caller_ptr)->s_ipc_to, nr_to_id(src_dst))) {
07538             kprintf("sys_call: ipc mask denied %d sending to %d\n",
07539                 proc_nr(caller_ptr), src_dst);
07540             return(ECALLDENIED);        /* call denied by ipc mask */
07541         }
07542
07543         if (isemptyn(src_dst) && !shutdown_started) {
07544             kprintf("sys_call: dead dest; %d, %d, %d\n",
07545                 function, proc_nr(caller_ptr), src_dst);
07546             return(EDEADDDST);         /* cannot send to the dead */
07547         }
07548     }
07549
07550     /* Now check if the call is known and try to perform the request. The only
07551     * system calls that exist in MINIX are sending and receiving messages.
07552     * - SENDREC: combines SEND and RECEIVE in a single system call
07553     * - SEND:    sender blocks until its message has been delivered
07554     * - RECEIVE: receiver blocks until an acceptable message has arrived
07555     * - NOTIFY:  nonblocking call; deliver notification or mark pending
07556     * - ECHO:    nonblocking call; directly echo back the message
07557     */
07558     switch(function) {
07559     case SENDREC:
07560         /* A flag is set so that notifications cannot interrupt SENDREC. */
07561         priv(caller_ptr)->s_flags |= SENDREC_BUSY;
07562         /* fall through */
07563     case SEND:
07564         result = mini_send(caller_ptr, src_dst, m_ptr, flags);
07565         if (function == SEND || result != OK) {
07566             break;                  /* done, or SEND failed */
07567         }                         /* fall through for SENDREC */
07568     case RECEIVE:

```

```

07569     if (function == RECEIVE)
0770         priv(caller_ptr)->s_flags &= ~SENDREC_BUSY;
07571     result = mini_receive(caller_ptr, src_dst, m_ptr, flags);
07572     break;
07573 case NOTIFY:
07574     result = mini_notify(caller_ptr, src_dst);
07575     break;
07576 case ECHO:
07577     CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);
07578     result = OK;
07579     break;
07580 default:
07581     result = EBADCALL;           /* illegal system call */
07582 }
07583
07584 /* Now, return the result of the system call to the caller. */
07585 return(result);
07586 }

07588 /*=====
07589 *          mini_send
07590 *=====
07591 PRIVATE int mini_send(caller_ptr, dst, m_ptr, flags)
07592 register struct proc *caller_ptr;      /* who is trying to send a message? */
07593 int dst;                            /* to whom is message being sent? */
07594 message *m_ptr;                    /* pointer to message buffer */
07595 unsigned flags;                   /* system call flags */
07596 {
07597 /* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
07598 * for this message, copy the message to it and unblock 'dst'. If 'dst' is
07599 * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
07600 */
07601 register struct proc *dst_ptr = proc_addr(dst);
07602 register struct proc **xpp;
07603 register struct proc *xp;
07604
07605 /* Check for deadlock by 'caller_ptr' and 'dst' sending to each other. */
07606 xp = dst_ptr;
07607 while (xp->p_rts_flags & SENDING) {           /* check while sending */
07608     xp = proc_addr(xp->p_sendto);            /* get xp's destination */
07609     if (xp == caller_ptr) return(ELOCKED); /* deadlock if cyclic */
07610 }
07611
07612 /* Check if 'dst' is blocked waiting for this message. The destination's
07613 * SENDING flag may be set when its SENDREC call blocked while sending.
07614 */
07615 if ((dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
07616     (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr)) {
07617     /* Destination is indeed waiting for this message. */
07618     CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
07619               dst_ptr->p_messbuf);
07620     if (((dst_ptr->p_rts_flags & ~RECEIVING) == 0) enqueue(dst_ptr);
07621 } else if ( !(flags & NON_BLOCKING) ) {
07622     /* Destination is not waiting. Block and dequeue caller. */
07623     caller_ptr->p_messbuf = m_ptr;
07624     if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
07625     caller_ptr->p_rts_flags |= SENDING;
07626     caller_ptr->p_sendto = dst;
07627
07628     /* Process is now blocked. Put in on the destination's queue. */
07629     xpp = &dst_ptr->p_caller_q;           /* find end of list */
07630     while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
07631     *xpp = caller_ptr;                  /* add caller to end */
07632     caller_ptr->p_q_link = NIL_PROC;    /* mark new end of list */
07633 } else {
07634     return(ENOTREADY);
07635 }

```

```

07636     return(OK);
07637 }

07639 /*=====
07640 *          mini_receive
07641 *=====
07642 PRIVATE int mini_receive(caller_ptr, src, m_ptr, flags)
07643 register struct proc *caller_ptr;           /* process trying to get message */
07644 int src;                                /* which message source is wanted */
07645 message *m_ptr;                          /* pointer to message buffer */
07646 unsigned flags;                         /* system call flags */
07647 {
07648 /* A process or task wants to get a message. If a message is already queued,
07649 * acquire it and deblock the sender. If no message from the desired source
07650 * is available block the caller, unless the flags don't allow blocking.
07651 */
07652 register struct proc **xpp;
07653 register struct notification **ntf_q_pp;
07654 message m;
07655 int bit_nr;
07656 sys_map_t *map;
07657 bitchunk_t *chunk;
07658 int i, src_id, src_proc_nr;
07659
07660 /* Check to see if a message from desired source is already available.
07661 * The caller's SENDING flag may be set if SENDREC couldn't send. If it is
07662 * set, the process should be blocked.
07663 */
07664 if (!(caller_ptr->p_rts_flags & SENDING)) {
07665
07666 /* Check if there are pending notifications, except for SENDREC. */
07667 if (!(priv(caller_ptr)->s_flags & SENDREC_BUSY)) {
07668
07669     map = &priv(caller_ptr)->s_notify_pending;
07670     for (chunk=&map->chunk[0]; chunk<&map->chunk[NR_SYS_CHUNKS]; chunk++) {
07671
07672         /* Find a pending notification from the requested source. */
07673         if (!*chunk) continue;                      /* no bits in chunk */
07674         for (i=0; !(*chunk & (1<<i)); ++i) {}      /* look up the bit */
07675         src_id = (chunk - &map->chunk[0]) * BITCHUNK_BITS + i;
07676         if (src_id >= NR_SYS_PROCS) break;        /* out of range */
07677         src_proc_nr = id_to_nr(src_id);            /* get source proc */
07678         if (src!=ANY && src!=src_proc_nr) continue; /* source not ok */
07679         *chunk &= ~(1 << i);                     /* no longer pending */
07680
07681         /* Found a suitable source, deliver the notification message. */
07682         BuildMess(&m, src_proc_nr, caller_ptr);       /* assemble message */
07683         CopyMess(src_proc_nr, proc_addr(HARDWARE), &m, caller_ptr, m_ptr);
07684         return(OK);                                /* report success */
07685     }
07686 }
07687
07688 /* Check caller queue. Use pointer pointers to keep code simple. */
07689 xpp = &caller_ptr->p_caller_q;
07690 while (*xpp != NIL_PROC) {
07691     if (src == ANY || src == proc_nr(*xpp)) {
07692         /* Found acceptable message. Copy it and update status. */
07693         CopyMess((*xpp)->p_nr, *xpp, (*xpp)->p_messbuf, caller_ptr, m_ptr);
07694         if (((*xpp)->p_rts_flags & ~SENDING) == 0) enqueue(*xpp);
07695         *xpp = (*xpp)->p_q_link;                  /* remove from queue */
07696         return(OK);                                /* report success */
07697     }
07698     xpp = &(*xpp)->p_q_link;                  /* proceed to next */
07699 }
07700 }
07701 */
07702 /* No suitable message is available or the caller couldn't send in SENDREC.

```

```

07703      * Block the process trying to receive, unless the flags tell otherwise.
07704      */
07705  if ( !(flags & NON_BLOCKING)) {
07706      caller_ptr->p_getfrom = src;
07707      caller_ptr->p_messbuf = m_ptr;
07708      if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
07709      caller_ptr->p_rts_flags |= RECEIVING;
07710      return(OK);
07711  } else {
07712      return(ENOTREADY);
07713  }
07714 }
07715 /*=====
07716          mini_notify
07717  =====*/
07718 PRIVATE int mini_notify(caller_ptr, dst)
07719 register struct proc *caller_ptr;           /* sender of the notification */
07720 int dst;                                /* which process to notify */
07721 {
07722     register struct proc *dst_ptr = proc_addr(dst);
07723     int src_id;                         /* source id for late delivery */
07724     message m;                          /* the notification message */
07725
07726     /* Check to see if target is blocked waiting for this message. A process
07727      * can be both sending and receiving during a SENDREC system call.
07728      */
07729     if ((dst_ptr->p_rts_flags & (RECEIVING|SENDING)) == RECEIVING &&
07730         ! (priv(dst_ptr)->s_flags & SENDREC_BUSY) &&
07731         (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr)) {
07732
07733         /* Destination is indeed waiting for a message. Assemble a notification
07734          * message and deliver it. Copy from pseudo-source HARDWARE, since the
07735          * message is in the kernel's address space.
07736          */
07737         BuildMess(&m, proc_nr(caller_ptr), dst_ptr);
07738         CopyMess(proc_nr(caller_ptr), proc_addr(HARDWARE), &m,
07739                   dst_ptr, dst_ptr->p_messbuf);
07740         dst_ptr->p_rts_flags &= ~RECEIVING;        /* deblock destination */
07741         if (dst_ptr->p_rts_flags == 0) enqueue(dst_ptr);
07742         return(OK);
07743     }
07744
07745     /* Destination is not ready to receive the notification. Add it to the
07746      * bit map with pending notifications. Note the indirectness: the system id
07747      * instead of the process number is used in the pending bit map.
07748      */
07749     src_id = priv(caller_ptr)->s_id;
07750     set_sys_bit(priv(dst_ptr)->s_notify_pending, src_id);
07751     return(OK);
07752 }
07753 }

07754 /*=====
07755          lock_notify
07756  =====*/
07757 PUBLIC int lock_notify(src, dst)
07758 int src;                      /* sender of the notification */
07759 int dst;                      /* who is to be notified */
07760 {
07761     /* Safe gateway to mini_notify() for tasks and interrupt handlers. The sender
07762      * is explicitly given to prevent confusion where the call comes from. MINIX
07763      * kernel is not reentrant, which means that interrupts are disabled after
07764      * the first kernel entry (hardware interrupt, trap, or exception). Locking
07765      * is done by temporarily disabling interrupts.
07766      */
07767     int result;
07768 }
```

```

07770 /* Exception or interrupt occurred, thus already locked. */
07771 if (k_reenter >= 0) {
07772     result = mini_notify(proc_addr(src), dst);
07773 }
07774
07775 /* Call from task level, locking is required. */
07776 else {
07777     lock(0, "notify");
07778     result = mini_notify(proc_addr(src), dst);
07779     unlock(0);
07780 }
07781 return(result);
07782 }

07784 /*=====
07785 *          enqueue
07786 *=====
07787 PRIVATE void enqueue(rp)
07788 register struct proc *rp;      /* this process is now runnable */
07789 {
07790 /* Add 'rp' to one of the queues of runnable processes. This function is
07791 * responsible for inserting a process into one of the scheduling queues.
07792 * The mechanism is implemented here. The actual scheduling policy is
07793 * defined in sched() and pick_proc().
07794 */
07795     int q;                      /* scheduling queue to use */
07796     int front;                  /* add to front or back */
07797
07798 /* Determine where to insert to process. */
07799 sched(rp, &q, &front);
07800
07801 /* Now add the process to the queue. */
07802 if (rdy_head[q] == NIL_PROC) {           /* add to empty queue */
07803     rdy_head[q] = rdy_tail[q] = rp;        /* create a new queue */
07804     rp->p_nextready = NIL_PROC;          /* mark new end */
07805 }
07806 else if (front) {                      /* add to head of queue */
07807     rp->p_nextready = rdy_head[q];       /* chain head of queue */
07808     rdy_head[q] = rp;                    /* set new queue head */
07809 }
07810 else {                                /* add to tail of queue */
07811     rdy_tail[q]->p_nextready = rp;       /* chain tail of queue */
07812     rdy_tail[q] = rp;                   /* set new queue tail */
07813     rp->p_nextready = NIL_PROC;          /* mark new end */
07814 }
07815
07816 /* Now select the next process to run. */
07817 pick_proc();
07818 }

07820 /*=====
07821 *          dequeue
07822 *=====
07823 PRIVATE void dequeue(rp)
07824 register struct proc *rp;      /* this process is no longer runnable */
07825 {
07826 /* A process must be removed from the scheduling queues, for example, because
07827 * it has blocked. If the currently active process is removed, a new process
07828 * is picked to run by calling pick_proc().
07829 */
07830     register int q = rp->p_priority;        /* queue to use */
07831     register struct proc **xpp;              /* iterate over queue */
07832     register struct proc *prev_xp;
07833
07834 /* Side-effect for kernel: check if the task's stack still is ok? */
07835 if (iskernelp(rp)) {

```

```

07836         if (*priv(rp)->s_stack_guard != STACK_GUARD)
07837             panic("stack overrun by task", proc_nr(rp));
07838     }
07839
07840     /* Now make sure that the process is not in its ready queue. Remove the
07841     * process if it is found. A process can be made unready even if it is not
07842     * running by being sent a signal that kills it.
07843     */
07844     prev_xp = NIL_PROC;
07845     for (xpp = &rdy_head[q]; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {
07846
07847         if (*xpp == rp) {                                /* found process to remove */
07848             *xpp = (*xpp)->p_nextready;                /* replace with next chain */
07849             if (rp == rdy_tail[q])                      /* queue tail removed */
07850                 rdy_tail[q] = prev_xp;                  /* set new tail */
07851             if (rp == proc_ptr || rp == next_ptr) /* active process removed */
07852                 pick_proc();                         /* pick new process to run */
07853             break;
07854         }
07855         prev_xp = *xpp;                                /* save previous in chain */
07856     }
07857 }

07858 /*=====
07859      *                               sched
07860      *=====
07861 */
07862 PRIVATE void sched(rp, queue, front)
07863 register struct proc *rp;                                /* process to be scheduled */
07864 int *queue;                                         /* return: queue to use */
07865 int *front;                                         /* return: front or back */
07866 {
07867     /* This function determines the scheduling policy. It is called whenever a
07868     * process must be added to one of the scheduling queues to decide where to
07869     * insert it. As a side-effect the process' priority may be updated.
07870     */
07871     static struct proc *prev_ptr = NIL_PROC;           /* previous without time */
07872     int time_left = (rp->p_ticks_left > 0);        /* quantum fully consumed */
07873     int penalty = 0;                                    /* change in priority */
07874
07875     /* Check whether the process has time left. Otherwise give a new quantum
07876     * and possibly raise the priority. Processes using multiple quanta
07877     * in a row get a lower priority to catch infinite loops in high priority
07878     * processes (system servers and drivers).
07879     */
07880     if (!time_left) {                                  /* quantum consumed ? */
07881         rp->p_ticks_left = rp->p_quantum_size;       /* give new quantum */
07882         if (prev_ptr == rp) penalty++;                 /* catch infinite loops */
07883         else penalty--;                            /* give slow way back */
07884         prev_ptr = rp;                           /* store ptr for next */
07885     }
07886
07887     /* Determine the new priority of this process. The bounds are determined
07888     * by IDLE's queue and the maximum priority of this process. Kernel tasks
07889     * and the idle process are never changed in priority.
07890     */
07891     if (penalty != 0 && !iskernelp(rp)) {
07892         rp->p_priority += penalty;                   /* update with penalty */
07893         if (rp->p_priority < rp->p_max_priority) /* check upper bound */
07894             rp->p_priority=rp->p_max_priority;
07895         else if (rp->p_priority > IDLE_Q-1)        /* check lower bound */
07896             rp->p_priority = IDLE_Q-1;
07897     }
07898
07899     /* If there is time left, the process is added to the front of its queue,
07900     * so that it can immediately run. The queue to use simply is always the
07901     * process' current priority.
07902     */

```

```

07903     *queue = rp->p_priority;
07904     *front = time_left;
07905 }
07906
07907 /*=====
07908 *          pick_proc
07909 *=====
07910 PRIVATE void pick_proc()
07911 {
07912 /* Decide who to run now. A new process is selected by setting 'next_ptr'.
07913 * When a billable process is selected, record it in 'bill_ptr', so that the
07914 * clock task can tell who to bill for system time.
07915 */
07916 register struct proc *rp;           /* process to run */
07917 int q;                           /* iterate over queues */
07918
07919 /* Check each of the scheduling queues for ready processes. The number of
07920 * queues is defined in proc.h, and priorities are set in the image table.
07921 * The lowest queue contains IDLE, which is always ready.
07922 */
07923 for (q=0; q < NR_SCHED_QUEUES; q++) {
07924     if ((rp = rdy_head[q]) != NIL_PROC) {
07925         next_ptr = rp;                  /* run process 'rp' next */
07926         if (priv(rp)->s_flags & BILLABLE)
07927             bill_ptr = rp;            /* bill for system time */
07928         return;
07929     }
07930 }
07931 }
07932
07933 /*=====
07934 *          lock_send
07935 *=====
07936 PUBLIC int lock_send(dst, m_ptr)
07937 int dst;                         /* to whom is message being sent? */
07938 message *m_ptr;                 /* pointer to message buffer */
07939 {
07940 /* Safe gateway to mini_send() for tasks. */
07941     int result;
07942     lock(2, "send");
07943     result = mini_send(proc_ptr, dst, m_ptr, NON_BLOCKING);
07944     unlock(2);
07945     return(result);
07946 }
07947
07948 /*=====
07949 *          lock_enqueue
07950 *=====
07951 PUBLIC void lock_enqueue(rp)
07952 struct proc *rp;                /* this process is now runnable */
07953 {
07954 /* Safe gateway to enqueue() for tasks. */
07955     lock(3, "enqueue");
07956     enqueue(rp);
07957     unlock(3);
07958 }
07959
07960 /*=====
07961 *          lock_dequeue
07962 *=====
07963 PUBLIC void lock_dequeue(rp)
07964 struct proc *rp;                /* this process is no longer runnable */
07965 {
07966 /* Safe gateway to dequeue() for tasks. */
07967     lock(4, "dequeue");
07968     dequeue(rp);
07969     unlock(4);
07970 }

```

```
+++++
kernel/exception.c
+++++

08000 /* This file contains a simple exception handler. Exceptions in user
08001 * processes are converted to signals. Exceptions in a kernel task cause
08002 * a panic.
08003 */
08004
08005 #include "kernel.h"
08006 #include <signal.h>
08007 #include "proc.h"
08008
08009 /*=====
08010 *           exception
08011 *=====
08012 PUBLIC void exception(vec_nr)
08013 unsigned vec_nr;
08014 {
08015 /* An exception or unexpected interrupt has occurred. */
08016
08017     struct ex_s {
08018         char *msg;
08019         int signum;
08020         int minprocessor;
08021     };
08022     static struct ex_s ex_data[] = {
08023         {"Divide error", SIGFPE, 86 },
08024         {"Debug exception", SIGTRAP, 86 },
08025         {"Nonmaskable interrupt", SIGBUS, 86 },
08026         {"Breakpoint", SIGEMT, 86 },
08027         {"Overflow", SIGFPE, 86 },
08028         {"Bounds check", SIGFPE, 186 },
08029         {"Invalid opcode", SIGILL, 186 },
08030         {"Coprocessor not available", SIGFPE, 186 },
08031         {"Double fault", SIGBUS, 286 },
08032         {"Copressor segment overrun", SIGSEGV, 286 },
08033         {"Invalid TSS", SIGSEGV, 286 },
08034         {"Segment not present", SIGSEGV, 286 },
08035         {"Stack exception", SIGSEGV, 286 }, /* STACK_FAULT already used */
08036         {"General protection", SIGSEGV, 286 },
08037         {"Page fault", SIGSEGV, 386 }, /* not close */
08038         {NIL_PTR, SIGILL, 0 }, /* probably software trap */
08039         {"Coprocessor error", SIGFPE, 386 },
08040     };
08041     register struct ex_s *ep;
08042     struct proc *saved_proc;
08043
08044     /* Save proc_ptr, because it may be changed by debug statements. */
08045     saved_proc = proc_ptr;
08046
08047     ep = &ex_data[vec_nr];
08048
08049     if (vec_nr == 2) { /* spurious NMI on some machines */
08050         kprintf("got spurious NMI\n");
08051         return;
08052     }
08053
08054     /* If an exception occurs while running a process, the k_reenter variable
08055     * will be zero. Exceptions in interrupt handlers or system traps will make
08056     * k_reenter larger than zero.
08057     */
08058     if (k_reenter == 0 && !iskernelp(saved_proc)) {
08059         cause_sig(proc_nr(saved_proc), ep->signum);
08060         return;
08061     }
08062 }
```

```

08063 /* Exception in system code. This is not supposed to happen. */
08064 if (ep->msg == NIL_PTR || machine.processor < ep->minprocessor)
08065     kprintf("\nIntel-reserved exception %d\n", vec_nr);
08066 else
08067     kprintf("\n%s\n", ep->msg);
08068 kprintf("k_reenter = %d ", k_reenter);
08069 kprintf("process %d (%s), ", proc_nr(saved_proc), saved_proc->p_name);
08070 kprintf("pc = %u:0x%lx", (unsigned) saved_proc->p_reg.cs,
08071     (unsigned) saved_proc->p_reg.pc);
08072
08073 panic("exception in a kernel task", NO_NUM);
08074 }

+++++-----+
kernel/i8259.c
+++++-----+

08100 /* This file contains routines for initializing the 8259 interrupt controller:
08101 *      put_irq_handler: register an interrupt handler
08102 *      rm_irq_handler: deregister an interrupt handler
08103 *      intr_handle:    handle a hardware interrupt
08104 *      intr_init:      initialize the interrupt controller(s)
08105 */
08106
08107 #include "kernel.h"
08108 #include "proc.h"
08109 #include <minix/com.h>
08110
08111 #define ICW1_AT      0x11 /* edge triggered, cascade, need ICW4 */
08112 #define ICW1_PC      0x13 /* edge triggered, no cascade, need ICW4 */
08113 #define ICW1_PS      0x19 /* level triggered, cascade, need ICW4 */
08114 #define ICW4_AT_SLAVE 0x01 /* not SFNM, not buffered, normal EOI, 8086 */
08115 #define ICW4_AT_MASTER 0x05 /* not SFNM, not buffered, normal EOI, 8086 */
08116 #define ICW4_PC_SLAVE 0x09 /* not SFNM, buffered, normal EOI, 8086 */
08117 #define ICW4_PC_MASTER 0x0D /* not SFNM, buffered, normal EOI, 8086 */
08118
08119 #define set_vec(nr, addr) ((void)0)
08120
08121 /*=====
08122 *          intr_init
08123 *=====
08124 PUBLIC void intr_init(mine)
08125 int mine;
08126 {
08127 /* Initialize the 8259s, finishing with all interrupts disabled. This is
08128 * only done in protected mode, in real mode we don't touch the 8259s, but
08129 * use the BIOS locations instead. The flag "mine" is set if the 8259s are
08130 * to be programmed for MINIX, or to be reset to what the BIOS expects.
08131 */
08132 int i;
08133
08134 intr_disable();
08135
08136 /* The AT and newer PS/2 have two interrupt controllers,'one master,
08137 * one slaved at IRQ 2. (We don't have to deal with the PC that
08138 * has just one controller, because it must run in real mode.)
08139 */
08140 outb(INT_CTL, machine.ps_mca ? ICW1_PS : ICW1_AT);
08141 outb(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
08142 outb(INT_CTLMASK, (1 << CASCADE_IRQ));           /* ICW2 for master */
08143 outb(INT_CTLMASK, ICW4_AT_MASTER);                  /* ICW3 tells slaves */
08144

```

```

08145     outb(INT_CTLMASK, ~(1 << CASCADE_IRQ));           /* IRQ 0-7 mask */
08146     outb(INT2_CTL, machine.ps.mca ? ICW1_PS : ICW1_AT);
08147     outb(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);
08148                                         /* ICW2 for slave */
08149     outb(INT2_CTLMASK, CASCADE_IRQ);           /* ICW3 is slave nr */
08150     outb(INT2_CTLMASK, ICW4_AT_SLAVE);
08151     outb(INT2_CTLMASK, '0');                  /* IRQ 8-15 mask */
08152
08153     /* Copy the BIOS vectors from the BIOS to the Minix location, so we
08154      * can still make BIOS calls without reprogramming the i8259s.
08155      */
08156     phys_copy(BIOS_VECTOR(0) * 4L, VECTOR(0) * 4L, 8 * 4L);
08157 }

08159 /*=====
08160 *          put_irq_handler
08161 *=====
08162 PUBLIC void put_irq_handler(hook, irq, handler)
08163 irq_hook_t *hook;
08164 int irq;
08165 irq_handler_t handler;
08166 {
08167 /* Register an interrupt handler. */
08168     int id;
08169     irq_hook_t **line;
08170
08171     if (irq < 0 || irq >= NR_IRQ_VECTORS)
08172         panic("invalid call to put_irq_handler", irq);
08173
08174     line = &irq_handlers[irq];
08175     id = 1;
08176     while (*line != NULL) {
08177         if (hook == *line) return;           /* extra initialization */
08178         line = &(*line)->next;
08179         id <= 1;
08180     }
08181     if (id == 0) panic("Too many handlers for irq", irq);
08182
08183     hook->next = NULL;
08184     hook->handler = handler;
08185     hook->irq = irq;
08186     hook->id = id;
08187     *line = hook;
08188
08189     irq_use |= 1 << irq;
08190 }

08192 /*=====
08193 *          rm_irq_handler
08194 *=====
08195 PUBLIC void rm_irq_handler(hook)
08196 irq_hook_t *hook;
08197 {
08198 /* Unregister an interrupt handler. */
08199     int irq = hook->irq;
08200     int id = hook->id;
08201     irq_hook_t **line;
08202
08203     if (irq < 0 || irq >= NR_IRQ_VECTORS)
08204         panic("invalid call to rm_irq_handler", irq);
08205
08206     line = &irq_handlers[irq];
08207     while (*line != NULL) {
08208         if ((*line)->id == id) {
08209             (*line) = (*line)->next;
08210             if (!irq_handlers[irq]) irq_use &= ~(1 << irq);
08211             return;

```

```

08212      }
08213      line = &(*line)->next;
08214    }
08215    /* When the handler is not found, normally return here. */
08216  }

08218 /*=====
0819 *           intr_handle
0820 *=====
0821 PUBLIC void intr_handle(hook)
0822 irq_hook_t *hook;
0823 {
0824 /* Call the interrupt handlers for an interrupt with the given hook list.
0825 * The assembly part of the handler has already masked the IRQ, reenabled the
0826 * controller(s) and enabled interrupts.
0827 */
0828
0829 /* Call list of handlers for an IRQ. */
0830 while (hook != NULL) {
0831     /* For each handler in the list, mark it active by setting its ID bit,
0832      * call the function, and unmark it if the function returns true.
0833      */
0834     irq_actids[hook->irq] |= hook->id;
0835     if ((*hook->handler)(hook)) irq_actids[hook->irq] &= ~hook->id;
0836     hook = hook->next;
0837 }
0838
0839 /* The assembly code will now disable interrupts, unmask the IRQ if and only
0840 * if all active ID bits are cleared, and restart a process.
0841 */
0842 }
```

```

+++++
kernel/protect.c
+++++

08300 /* This file contains code for initialization of protected mode, to initialize
08301   * code and data segment descriptors, and to initialize global descriptors
08302   * for local descriptors in the process table.
08303 */
08304
08305 #include "kernel.h"
08306 #include "proc.h"
08307 #include "protect.h"
08308
08309 #define INT_GATE_TYPE  (INT_286_GATE | DESC_386_BIT)
08310 #define TSS_TYPE        (AVL_286_TSS | DESC_386_BIT)
08311
08312 struct descableptr_s {
08313     char limit[sizeof(u16_t)];
08314     char base[sizeof(u32_t)];           /* really u24_t + pad for 286 */
08315 };
08316
08317 struct gatedesc_s {
08318     u16_t offset_low;
08319     u16_t selector;
08320     u8_t pad;                      /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
08321     u8_t p_dpl_type;               /* |P|DL|0|TYPE| */
08322     u16_t offset_high;
08323 };
08324
08325 struct tss_s {
08326     reg_t backlink;
08327     reg_t sp0;                     /* stack pointer to use during interrupt */
08328     reg_t ss0;                    /* " " " " " " */

```

```

08329    reg_t sp1;
08330    reg_t ss1;
08331    reg_t sp2;
08332    reg_t ss2;
08333    reg_t cr3;
08334    reg_t ip;
08335    reg_t flags;
08336    reg_t ax;
08337    reg_t cx;
08338    reg_t dx;
08339    reg_t bx;
08340    reg_t sp;
08341    reg_t bp;
08342    reg_t si;
08343    reg_t di;
08344    reg_t es;
08345    reg_t cs;
08346    reg_t ss;
08347    reg_t ds;
08348    reg_t fs;
08349    reg_t gs;
08350    reg_t ldt;
08351    u16_t trap;
08352    u16_t iobase;
08353 /* u8_t iomap[0]; */
08354 };
08355
08356 PUBLIC struct segdesc_s gdt[GDT_SIZE];           /* used in klib.s and mpx.s */
08357 PRIVATE struct gatedesc_s idt[IDT_SIZE];          /* zero-init so none present */
08358 PUBLIC struct tss_s tss;                          /* zero init */
08359
08360 FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, vir_bytes offset,
08361                      unsigned dpl_type) );
08362 FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,
08363                      vir_bytes size) );
08364
08365 /*=====
08366      *                         prot_init
08367      *=====
08368 */
08369 PUBLIC void prot_init()
08370 {
08371     /* Set up tables for protected mode.
08372      * All GDT slots are allocated at compile time.
08373      */
08374     struct gate_table_s *gtp;
08375     struct descableptr_s *dtp;
08376     unsigned ldt_index;
08377     register struct proc *rp;
08378
08379     static struct gate_table_s {
08380         _PROTOTYPE( void (*gate), (void) );
08381         unsigned char vec_nr;
08382         unsigned char privilege;
08383     }
08384     gate_table[] = {
08385         { divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE },
08386         { single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE },
08387         { nmi, NMI_VECTOR, INTR_PRIVILEGE },
08388         { breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE },
08389         { overflow, OVERFLOW_VECTOR, USER_PRIVILEGE },
08390         { bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE },
08391         { inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE },
08392         { copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE },
08393         { double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE },
08394         { copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE },
08395         { inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE },
08396         { segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE },

```

```

08396     { stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE },
08397     { general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE },
08398     { page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE },
08399     { copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE },
08400     { hwint00, VECTOR( 0), INTR_PRIVILEGE },
08401     { hwint01, VECTOR( 1), INTR_PRIVILEGE },
08402     { hwint02, VECTOR( 2), INTR_PRIVILEGE },
08403     { hwint03, VECTOR( 3), INTR_PRIVILEGE },
08404     { hwint04, VECTOR( 4), INTR_PRIVILEGE },
08405     { hwint05, VECTOR( 5), INTR_PRIVILEGE },
08406     { hwint06, VECTOR( 6), INTR_PRIVILEGE },
08407     { hwint07, VECTOR( 7), INTR_PRIVILEGE },
08408     { hwint08, VECTOR( 8), INTR_PRIVILEGE },
08409     { hwint09, VECTOR( 9), INTR_PRIVILEGE },
08410     { hwint10, VECTOR(10), INTR_PRIVILEGE },
08411     { hwint11, VECTOR(11), INTR_PRIVILEGE },
08412     { hwint12, VECTOR(12), INTR_PRIVILEGE },
08413     { hwint13, VECTOR(13), INTR_PRIVILEGE },
08414     { hwint14, VECTOR(14), INTR_PRIVILEGE },
08415     { hwint15, VECTOR(15), INTR_PRIVILEGE },
08416     { s_call, SYS386_VECTOR, USER_PRIVILEGE },      /* 386 system call */
08417     { level0_call, LEVEL0_VECTOR, TASK_PRIVILEGE },
08418 };
08419
08420 /* Build gdt and idt pointers in GDT where the BIOS expects them. */
08421 dtp= (struct descableptr_s *) &gdt[GDT_INDEX];
08422 * (u16_t *) dtp->limit = (sizeof gdt) - 1;
08423 * (u32_t *) dtp->base = vir2phys(gdt);
08424
08425 dtp= (struct descableptr_s *) &gdt[IDT_INDEX];
08426 * (u16_t *) dtp->limit = (sizeof idt) - 1;
08427 * (u32_t *) dtp->base = vir2phys(idt);
08428
08429 /* Build segment descriptors for tasks and interrupt handlers. */
08430 init_codeseg(&gdt[CS_INDEX],
08431     kinfo.code_base, kinfo.code_size, INTR_PRIVILEGE);
08432 init_dataseg(&gdt[DS_INDEX],
08433     kinfo.data_base, kinfo.data_size, INTR_PRIVILEGE);
08434 init_dataseg(&gdt[ES_INDEX], 0L, 0, TASK_PRIVILEGE);
08435
08436 /* Build scratch descriptors for functions in klib88. */
08437 init_dataseg(&gdt[DS_286_INDEX], 0L, 0, TASK_PRIVILEGE);
08438 init_dataseg(&gdt[ES_286_INDEX], 0L, 0, TASK_PRIVILEGE);
08439
08440 /* Build local descriptors in GDT for LDT's in process table.
08441 * The LDT's are allocated at compile time in the process table, and
08442 * initialized whenever a process' map is initialized or changed.
08443 */
08444 for (rp = BEG_PROC_ADDR, ldt_index = FIRST_LDT_INDEX;
08445     rp < END_PROC_ADDR; ++rp, ldt_index++) {
08446     init_dataseg(&gdt[ldt_index], vir2phys(rp->p_ldt),
08447         sizeof(rp->p_ldt), INTR_PRIVILEGE);
08448     gdt[ldt_index].access = PRESENT | LDT;
08449     rp->p_ldt_sel = ldt_index * DESC_SIZE;
08450 }
08451
08452 /* Build main TSS.
08453 * This is used only to record the stack pointer to be used after an
08454 * interrupt.
08455 * The pointer is set up so that an interrupt automatically saves the
08456 * current process's registers ip:cs:f:sp:ss in the correct slots in the
08457 * process table.
08458 */
08459 tss.ss0 = DS_SELECTOR;

```

```

08460     init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), sizeof(tss), INTR_PRIVILEGE);
08461     gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;
08462
08463     /* Build descriptors for interrupt gates in IDT. */
08464     for (gtp = &gate_table[0];
08465         gtp < &gate_table[sizeof(gate_table) / sizeof(gate_table[0]); ++gtp) {
08466         int_gate(gtp->vec_nr, (vir_bytes) gtp->gate,
08467                  PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
08468     }
08469
08470     /* Complete building of main TSS. */
08471     tss.iobase = sizeof(tss);      /* empty i/o permissions map */
08472 }

08474 /*=====
08475 *          init_codeseg
08476 *=====
08477 PUBLIC void init_codeseg(segdp, base, size, privilege)
08478 register struct segdesc_s *segdp;
08479 phys_bytes base;
08480 vir_bytes size;
08481 int privilege;
08482 {
08483     /* Build descriptor for a code segment. */
08484     sdesc(segdp, base, size);
08485     segdp->access = (privilege << DPL_SHIFT)
08486             | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
08487             /* CONFORMING = 0, ACCESSED = 0 */
08488 }

08490 /*=====
08491 *          init_dataseg
08492 *=====
08493 PUBLIC void init_dataseg(segdp, base, size, privilege)
08494 register struct segdesc_s *segdp;
08495 phys_bytes base;
08496 vir_bytes size;
08497 int privilege;
08498 {
08499     /* Build descriptor for a data segment. */
08500     sdesc(segdp, base, size);
08501     segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
08502             /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
08503 }

08505 /*=====
08506 *          sdesc
08507 *=====
08508 PRIVATE void sdesc(segdp, base, size)
08509 register struct segdesc_s *segdp;
08510 phys_bytes base;
08511 vir_bytes size;
08512 {
08513     /* Fill in the size fields (base, limit and granularity) of a descriptor. */
08514     segdp->base_low = base;
08515     segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
08516     segdp->base_high = base >> BASE_HIGH_SHIFT;
08517
08518     --size;                                /* convert to a limit, 0 size means 4G */
08519     if (size > BYTE_GRAN_MAX) {
08520         segdp->limit_low = size >> PAGE_GRAN_SHIFT;
08521         segdp->granularity = GRANULAR | (size >>
08522                                         (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
08523     } else {
08524         segdp->limit_low = size;
08525         segdp->granularity = size >> GRANULARITY_SHIFT;
08526     }

```

```

08527     segdp->granularity |= DEFAULT;           /* means BIG for data seg */
08528 }

08530 /*=====
08531 *          seg2phys
08532 *=====
08533 PUBLIC phys_bytes seg2phys(seg)
08534 u16_t seg;
08535 {
08536 /* Return the base address of a segment, with seg being either a 8086 segment
08537 * register, or a 286/386 segment selector.
08538 */
08539     phys_bytes base;
08540     struct segdesc_s *segdp;
08541
08542     if (! machine.protected) {
08543         base = hclick_to_physb(seg);
08544     } else {
08545         segdp = &gdt[seg >> 3];
08546         base = ((u32_t) segdp->base_low << 0)
08547             | ((u32_t) segdp->base_middle << 16)
08548             | ((u32_t) segdp->base_high << 24);
08549     }
08550     return base;
08551 }

08553 /*=====
08554 *          phys2seg
08555 *=====
08556 PUBLIC void phys2seg(seg, off, phys)
08557 u16_t *seg;
08558 vir_bytes *off;
08559 phys_bytes phys;
08560 {
08561 /* Return a segment selector and offset that can be used to reach a physical
08562 * address, for use by a driver doing memory I/O in the A0000 ~ DFFFF range.
08563 */
08564     *seg = FLAT_DS_SELECTOR;
08565     *off = phys;
08566 }

08568 /*=====
08569 *          int_gate
08570 *=====
08571 PRIVATE void int_gate(vec_nr, offset, dpl_type)
08572 unsigned vec_nr;
08573 vir_bytes offset;
08574 unsigned dpl_type;
08575 {
08576 /* Build descriptor for an interrupt gate. */
08577     register struct gatedesc_s *idp;
08578
08579     idp = &idt[vec_nr];
08580     idp->offset_low = offset;
08581     idp->selector = CS_SELECTOR;
08582     idp->p_dpl_type = dpl_type;
08583     idp->offset_high = offset >> OFFSET_HIGH_SHIFT;
08584 }

08586 /*=====
08587 *          enable_iop
08588 *=====
08589 PUBLIC void enable_iop(pp)
08590 struct proc *pp;
08591 {
08592 /* Allow a user process to use I/O instructions. Change the I/O Permission
08593 * Level bits in the psw. These specify least-privileged Current Permission

```

```

08594 * Level allowed to execute I/O instructions. Users and servers have CPL 3.
08595 * You can't have less privilege than that. Kernel has CPL 0, tasks CPL 1.
08596 */
08597 pp->p_reg.psw |= 0x3000;
08598 }

08600 /*=====
08601 *          alloc_segments
08602 *=====
08603 PUBLIC void alloc_segments(rp)
08604 register struct proc *rp;
08605 {
08606 /* This is called at system initialization from main() and by do_newmap().
08607 * The code has a separate function because of all hardware-dependencies.
08608 * Note that IDLE is part of the kernel and gets TASK_PRIVILEGE here.
08609 */
08610 phys_bytes code_bytes;
08611 phys_bytes data_bytes;
08612 int privilege;
08613
08614 if (machine.protected) {
08615     data_bytes = (phys_bytes) (rp->p_memmap[S].mem_vir +
08616         rp->p_memmap[S].mem_len) << CLICK_SHIFT;
08617     if (rp->p_memmap[T].mem_len == 0)
08618         code_bytes = data_bytes; /* common I&D, poor protect */
08619     else
08620         code_bytes = (phys_bytes) rp->p_memmap[T].mem_len << CLICK_SHIFT;
08621     privilege = (iskernelp(rp)) ? TASK_PRIVILEGE : USER_PRIVILEGE;
08622     init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
08623         (phys_bytes) rp->p_memmap[T].mem_phys << CLICK_SHIFT,
08624         code_bytes, privilege);
08625     init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
08626         (phys_bytes) rp->p_memmap[D].mem_phys << CLICK_SHIFT,
08627         data_bytes, privilege);
08628     rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
08629     rp->p_reg.gs =
08630     rp->p_reg.fs =
08631     rp->p_reg.ss =
08632     rp->p_reg.es =
08633     rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
08634 } else {
08635     rp->p_reg.cs = click_to_hclick(rp->p_memmap[T].mem_phys);
08636     rp->p_reg.ss =
08637     rp->p_reg.es =
08638     rp->p_reg.ds = click_to_hclick(rp->p_memmap[D].mem_phys);
08639 }
08640 }
```

=====
kernel/klib.s
=====

```

08700 #
08701 ! Chooses between the 8086 and 386 versions of the low level kernel code.
08702
08703 #include <minix/config.h>
08704 #if _WORD_SIZE == 2
08705 #include "klib88.s"
08706 #else
08707 #include "klib386.s"
08708 #endif
```

```
=====
kernel/klib386.s
=====

08800  #
08801 ! sections
08802
08803 .sect .text; .sect .rom; .sect .data; .sect .bss
08804
08805 #include <minix/config.h>
08806 #include <minix/const.h>
08807 #include "const.h"
08808 #include "sconst.h"
08809 #include "protect.h"
08810
08811 ! This file contains a number of assembly code utility routines needed by the
08812 ! kernel. They are:
08813
08814 .define _monitor      ! exit Minix and return to the monitor
08815 .define _int86         ! let the monitor make an 8086 interrupt call
08816 .define _cp_mess       ! copies messages from source to destination
08817 .define _exit          ! dummy for library routines
08818 .define __exit         ! dummy for library routines
08819 .define __exit         ! dummy for library routines
08820 .define __main          ! dummy for GCC
08821 .define _phys_insw     ! transfer data from (disk controller) port to memory
08822 .define _phys_insb     ! likewise byte by byte
08823 .define _phys_outsw    ! transfer data from memory to (disk controller) port
08824 .define _phys_outsb    ! likewise byte by byte
08825 .define _enable_irq     ! enable an irq at the 8259 controller
08826 .define _disable_irq    ! disable an irq
08827 .define _phys_copy      ! copy data from anywhere to anywhere in memory
08828 .define _phys_memset    ! write pattern anywhere in memory
08829 .define _mem_rdw        ! copy one word from [segment:offset]
08830 .define _reset          ! reset the system
08831 .define _idle_task      ! task executed when there is no work
08832 .define _level0          ! call a function at level 0
08833 .define _read_tsc        ! read the cycle counter (Pentium and up)
08834 .define _read_cpu_flags  ! read the cpu flags
08835
08836 ! The routines only guarantee to preserve the registers the C compiler
08837 ! expects to be preserved (ebx, esi, edi, ebp, esp, segment registers, and
08838 ! direction bit in the flags).
08839
08840 .sect .text
08841 !*=====
08842 !*                      monitor                   *
08843 !*=====
08844 ! PUBLIC void monitor();
08845 ! Return to the monitor.
08846
08847 _monitor:
08848     mov    esp, (_mon_sp)           ! restore monitor stack pointer
08849     o16   mov    dx, SS_SELECTOR    ! monitor data segment
08850     mov    ds, dx
08851     mov    es, dx
08852     mov    fs, dx
08853     mov    gs, dx
08854     mov    ss, dx
08855     pop    edi
08856     pop    esi
08857     pop    ebp
08858     o16  retf                  ! return to the monitor
08859
08860
08861 !*=====
08862 !*                      int86                   *
08863 !*=====
```

```

08864 ! PUBLIC void int86();
08865 _int86:
08866     cmpb  (_mon_return), 0      ! is the monitor there?
08867     jnz   0f
08868     movb  ah, 0x01              ! an int 13 error seems appropriate
08869     movb  (_reg86+ 0), ah        ! reg86.w.f = 1 (set carry flag)
08870     movb  (_reg86+13), ah       ! reg86.b.ah = 0x01 = "invalid command"
08871     ret
08872 0:    push   ebp             ! save C registers
08873     push   esi
08874     push   edi
08875     push   ebx
08876     pushf
08877     cli                ! save flags
08878                           ! no interruptions
08879     inb    INT2_CTLMASK
08880     movb  ah, a1
08881     inb    INT_CTLMASK
08882     push   eax             ! save interrupt masks
08883     mov    eax, (_irq_use)    ! map of in-use IRQ's
08884     and   eax, ^[1<<CLOCK_IRQ] ! keep the clock ticking
08885     outb  INT_CTLMASK       ! enable all unused IRQ's and vv.
08886     movb  al, ah
08887     outb  INT2_CTLMASK
08888
08889     mov    eax, SS_SELECTOR  ! monitor data segment
08890     mov    ss, ax
08891     xchg  esp, (_mon_sp)      ! switch stacks
08892     push  (_reg86+36)         ! parameters used in INT call
08893     push  (_reg86+32)
08894     push  (_reg86+28)
08895     push  (_reg86+24)
08896     push  (_reg86+20)
08897     push  (_reg86+16)
08898     push  (_reg86+12)
08899     push  (_reg86+ 8)
08900     push  (_reg86+ 4)
08901     push  (_reg86+ 0)
08902     mov    ds, ax             ! remaining data selectors
08903     mov    es, ax
08904     mov    fs, ax
08905     mov    gs, ax
08906     push  cs
08907     push  return            ! kernel return address and selector
08908     o16 jmpf 20+2*4+10*4+2*4(esp) ! make the call
08909 return:
08910     pop   (_reg86+ 0)
08911     pop   (_reg86+ 4)
08912     pop   (_reg86+ 8)
08913     pop   (_reg86+12)
08914     pop   (_reg86+16)
08915     pop   (_reg86+20)
08916     pop   (_reg86+24)
08917     pop   (_reg86+28)
08918     pop   (_reg86+32)
08919     pop   (_reg86+36)
08920     lgdt  (_gdt+GDT_SELECTOR) ! reload global descriptor table
08921     jmpf  CS_SELECTOR:csinit ! restore everything
08922 csinit: mov   eax, DS_SELECTOR
08923     mov   ds, ax
08924     mov   es, ax
08925     mov   fs, ax
08926     mov   gs, ax
08927     mov   ss, ax
08928     xchg  esp, (_mon_sp)      ! unswitch stacks
08929     lidt  (_gdt+IDT_SELECTOR) ! reload interrupt descriptor table
08930     andb  (_gdt+TSS_SELECTOR+DESC_ACCESS), ~0x02 ! clear TSS busy bit

```

```

08931      mov    eax, TSS_SELECTOR
08932      ltr    ax                      ! set TSS register
08933
08934      pop    eax
08935      outb   INT_CTLMASK            ! restore interrupt masks
08936      movb   al, ah
08937      outb   INT2_CTLMASK
08938
08939      add    (_lost_ticks), ecx     ! record lost clock ticks
08940
08941      popf   eax                  ! restore flags
08942      pop    ebx                  ! restore C registers
08943      pop    edi
08944      pop    esi
08945      pop    ebp
08946      ret
08947
08948
08949      !***** cp_mess *****
08950      !*
08951      !***** cp_mess *****
08952      ! PUBLIC void cp_mess(int src, phys_ticks src_ticks, vir_bytes src_offset,
08953      !                                phys_ticks dst_ticks, vir_bytes dst_offset);
08954      ! This routine makes a fast copy of a message from anywhere in the address
08955      ! space to anywhere else. It also copies the source address provided as a
08956      ! parameter to the call into the first word of the destination message.
08957      !
08958      ! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
08959      ! correctly. Changing the definition of message in the type file and not
08960      ! changing it here will lead to total disaster.
08961
08962      CM_ARGS =      4 + 4 + 4 + 4 + 4      ! 4 + 4 + 4 + 4 + 4
08963      !           es ds edi esi eip      proc scl sof dcl dof
08964
08965      .align 16
08966      _cp_mess:
08967      cld
08968      push   esi
08969      push   edi
08970      push   ds
08971      push   es
08972
08973      mov    eax, FLAT_DS_SELECTOR
08974      mov    ds, ax
08975      mov    es, ax
08976
08977      mov    esi, CM_ARGS+4(esp)        ! src clicks
08978      shl    esi, CLICK_SHIFT
08979      add    esi, CM_ARGS+4+4(esp)      ! src offset
08980      mov    edi, CM_ARGS+4+4+4(esp)    ! dst clicks
08981      shl    edi, CLICK_SHIFT
08982      add    edi, CM_ARGS+4+4+4+4(esp)  ! dst offset
08983
08984      mov    eax, CM_ARGS(esp)         ! process number of sender
08985      stos   ! copy number of sender to dest message
08986      add    esi, 4                  ! do not copy first word
08987      mov    ecx, Msize - 1          ! remember, first word does not count
08988      rep
08989      movs   ! copy the message
08990
08991      pop    es
08992      pop    ds
08993      pop    edi
08994      pop    esi
08995      ret
08996
08997      ! that is all folks!

```

```

08998 !***** exit *****
08999 !*          exit
09000 !*****
09001 ! PUBLIC void exit();
09002 ! Some library routines use exit, so provide a dummy version.
09003 ! Actual calls to exit cannot occur in the kernel.
09004 ! GNU CC likes to call __main from main() for nonobvious reasons.
09005
09006 _exit:
09007 __exit:
09008 __exit:
09009     sti
09010     jmp    __exit
09011
09012 __main:
09013     ret
09014
09015
09016 !***** phys_insw *****
09017 !*          phys_insw
09018 !*****
09019 ! PUBLIC void phys_insw(Port_t port, phys_bytes buf, size_t count);
09020 ! Input an array from an I/O port. Absolute address version of insw().
09021
09022 __phys_insw:
09023     push   ebp
09024     mov    ebp, esp
09025     cld
09026     push   edi
09027     push   es
09028     mov    ecx, FLAT_DS_SELECTOR
09029     mov    es, cx
09030     mov    edx, 8(epb)           ! port to read from
09031     mov    edi, 12(epb)         ! destination addr
09032     mov    ecx, 16(epb)         ! byte count
09033     shr    ecx, 1              ! word count
09034     rep   ol6 ins            ! input many words
09035     pop    es
09036     pop    edi
09037     pop    ebp
09038     ret
09039
09040
09041 !***** phys_insb *****
09042 !*          phys_insb
09043 !*****
09044 ! PUBLIC void phys_insb(Port_t port, phys_bytes buf, size_t count);
09045 ! Input an array from an I/O port. Absolute address version of insb().
09046
09047 __phys_insb:
09048     push   ebp
09049     mov    ebp, esp
09050     cld
09051     push   edi
09052     push   es
09053     mov    ecx, FLAT_DS_SELECTOR
09054     mov    es, cx
09055     mov    edx, 8(epb)           ! port to read from
09056     mov    edi, 12(epb)         ! destination addr
09057     mov    ecx, 16(epb)         ! byte count
09058     shr    ecx, 1              ! word count
09059     rep   insb              ! input many bytes
09060     pop    es
09061     pop    edi
09062     pop    ebp
09063     ret
09064

```

```

09065
09066 !*-----*
09067 !*          phys_outsw
09068 !*-----*
09069 ! PUBLIC void phys_outsw(Port_t port, phys_bytes buf, size_t count);
09070 ! Output an array to an I/O port. Absolute address version of outsw().
09071
09072     .align 16
09073 _phys_outsw:
09074     push  ebp
09075     mov   ebp, esp
09076     cld
09077     push  esi
09078     push  ds
09079     mov   ecx, FLAT_DS_SELECTOR
09080     mov   ds, cx
09081     mov   edx, 8(ebp)           ! port to write to
09082     mov   esi, 12(ebp)          ! source addr
09083     mov   ecx, 16(ebp)          ! byte count
09084     shr   ecx, 1              ! word count
09085     rep  o16 outs            ! output many words
09086     pop   ds
09087     pop   esi
09088     pop   ebp
09089     ret
09090
09091
09092 !*-----*
09093 !*          phys_outsb
09094 !*-----*
09095 ! PUBLIC void phys_outsb(Port_t port, phys_bytes buf, size_t count);
09096 ! Output an array to an I/O port. Absolute address version of outsb().
09097
09098     .align 16
09099 _phys_outsb:
09100     push  ebp
09101     mov   ebp, esp
09102     cld
09103     push  esi
09104     push  ds
09105     mov   ecx, FLAT_DS_SELECTOR
09106     mov   ds, cx
09107     mov   edx, 8(ebp)           ! port to write to
09108     mov   esi, 12(ebp)          ! source addr
09109     mov   ecx, 16(ebp)          ! byte count
09110     rep  outsb               ! output many bytes
09111     pop   ds
09112     pop   esi
09113     pop   ebp
09114     ret
09115
09116
09117 !*-----*
09118 !*          enable_irq
09119 !*-----*/
09120 ! PUBLIC void enable_irq(irq_hook_t *hook)
09121 ! Enable an interrupt request line by clearing an 8259 bit.
09122 ! Equivalent C code for hook->irq < 8:
09123 !   if ((irq_actids[hook->irq] & ~hook->id) == 0)
09124 !     outb(INT_CTLMASK, inb(INT_CTLMASK) & ~(1 << irq));
09125
09126     .align 16
09127 _enable_irq:
09128     push  ebp
09129     mov   ebp, esp
09130     pushf

```

```

09131      cli
09132      mov    eax, 8(ebp)          ! hook
09133      mov    ecx, 8(eax)        ! irq
09134      mov    eax, 12(eax)       ! id bit
09135      not    eax
09136      and    _irq_actids(ecx*4), eax ! clear this id bit
09137      jnz    en_done           ! still masked by other handlers?
09138      movb   ah, 1
09139      rolb   ah, cl            ! ah = ~(1 << (irq % 8))
09140      mov    edx, INT_CTLMASK  ! enable irq < 8 at the master 8259
09141      cmpb   cl, 8
09142      jb    0f
09143      mov    edx, INT2_CTLMASK ! enable irq >= 8 at the slave 8259
09144 0:     inb    dx
09145      andb   al, ah
09146      outb   dx               ! clear bit at the 8259
09147  en_done: popf
09148      leave
09149      ret
09150
09151
09152  !*** disable_irq
09153  !*
09154  !*
09155  ! PUBLIC int disable_irq(irq_hook_t *hook)
09156  ! Disable an interrupt request line by setting an 8259 bit.
09157  ! Equivalent C code for irq < 8:
09158  !     irq_actids[hook->irq] |= hook->id;
09159  !     outb(INT_CTLMASK, inb(INT_CTLMASK) | (1 << irq));
09160  ! Returns true iff the interrupt was not already disabled.
09161
09162      .align 16
09163  _disable_irq:
09164      push   ebp
09165      mov    ebp, esp
09166      pushf
09167      cli
09168      mov    eax, 8(ebp)          ! hook
09169      mov    ecx, 8(eax)        ! irq
09170      mov    eax, 12(eax)       ! id bit
09171      or     _irq_actids(ecx*4), eax ! set this id bit
09172      movb   ah, 1
09173      rolb   ah, cl            ! ah = (1 << (irq % 8))
09174      mov    edx, INT_CTLMASK  ! disable irq < 8 at the master 8259
09175      cmpb   cl, 8
09176      jb    0f
09177      mov    edx, INT2_CTLMASK ! disable irq >= 8 at the slave 8259
09178 0:     inb    dx
09179      testb  al, ah
09180      jnz    dis_already       ! already disabled?
09181      orb    al, ah
09182      outb   dx               ! set bit at the 8259
09183      mov    eax, 1             ! disabled by this function
09184      popf
09185      leave
09186      ret
09187  dis_already:
09188      xor    eax, eax          ! already disabled
09189      popf
09190      leave
09191      ret
09192
09193
09194  !*** phys_copy
09195  !*
09196  !*
09197  ! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
09198  !                         phys_bytes bytecount);

```

```

09199 ! Copy a block of physical memory.
09200
09201 PC_ARGS =      4 + 4 + 4 + 4    ! 4 + 4 + 4
09202 !           es edi esi eip   src dst len
09203
09204     .align 16
09205 _phys_copy:
09206     cld
09207     push  esi
09208     push  edi
09209     push  es
09210
09211     mov   eax, FLAT_DS_SELECTOR
09212     mov   es, ax
09213
09214     mov   esi, PC_ARGS(esp)
09215     mov   edi, PC_ARGS+4(esp)
09216     mov   eax, PC_ARGS+4+4(esp)
09217
09218     cmp   eax, 10          ! avoid align overhead for small counts
09219     jb    pc_small
09220     mov   ecx, esi          ! align source, hope target is too
09221     neg   ecx
09222     and  ecx, 3            ! count for alignment
09223     sub   eax, ecx
09224     rep
09225     eseg movsb
09226     mov   ecx, eax
09227     shr   ecx, 2            ! count of dwords
09228     rep
09229     eseg movs
09230     and  eax, 3
09231     pc_small:
09232     xchg  ecx, eax          ! remainder
09233     rep
09234     eseg movsb
09235
09236     pop   es
09237     pop   edi
09238     pop   esi
09239     ret
09240
09241 !=====
09242 !*          phys_memset          *
09243 !=====
09244 ! PUBLIC void phys_memset(phys_bytes source, unsigned long pattern,
09245 !     phys_bytes bytecount);
09246 ! Fill a block of physical memory with pattern.
09247
09248     .align 16
09249 _phys_memset:
09250     push  ebp
09251     mov   ebp, esp
09252     push  esi
09253     push  ebx
09254     push  ds
09255     mov   esi, 8(epb)
09256     mov   eax, 16(epb)
09257     mov   ebx, FLAT_DS_SELECTOR
09258     mov   ds, bx
09259     mov   ebx, 12(epb)
09260     shr   eax, 2
09261     fill_start:
09262     mov   (esi), ebx
09263     add   esi, 4
09264     dec   eax
09265     jnz   fill_start

```

```

09266      ! Any remaining bytes?
09267      mov    eax, 16(ebp)
09268      and    eax, 3
09269  remain_fill:
09270      cmp    eax, 0
09271      jz     fill_done
09272      movb   bl, 12(ebp)
09273      movb   (esi), bl
09274      add    esi, 1
09275      inc    ebp
09276      dec    eax
09277      jmp    remain_fill
09278  fill_done:
09279      pop    ds
09280      pop    ebx
09281      pop    esi
09282      pop    ebp
09283      ret
09284
09285  !*=====
09286  !*          mem_rdw
09287  !*=====
09288  ! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);
09289  ! Load and return word at far pointer segment:offset.
09290
09291      .align 16
09292 _mem_rdw:
09293      mov    cx, ds
09294      mov    ds, 4(esp)           ! segment
09295      mov    eax, 4+4(esp)        ! offset
09296      movzx  eax, (eax)          ! word to return
09297      mov    ds, cx
09298      ret
09299
09300
09301  !*=====
09302  !*          reset
09303  !*=====
09304  ! PUBLIC void reset();
09305  ! Reset the system by loading IDT with offset 0 and interrupting.
09306
09307 _reset:
09308      lidt   (idt_zero)
09309      int    3                  ! anything goes, the 386 will not like it
09310 .sect .data
09311 idt_zero:   .data4 0, 0
09312 .sect .text
09313
09314
09315  !*=====
09316  !*          idle_task
09317  !*=====
09318 _idle_task:
09319  ! This task is called when the system has nothing else to do. The HLT
09320  ! instruction puts the processor in a state where it draws minimum power.
09321      push   halt
09322      call   _level0           ! level0(halt)
09323      pop    eax
09324      jmp    _idle_task
09325 halt:
09326      sti
09327      hlt
09328      cli
09329      ret
09330
09331  !*=====
09332  !*          level0
09333  !*=====

```

```

09334 ! PUBLIC void level0(void (*func)(void))
09335 ! Call a function at permission level 0. This allows kernel tasks to do
09336 ! things that are only possible at the most privileged CPU level.
09337 !
09338 _level0:
09339     mov    eax, 4(esp)
09340     mov    (_level0_func), eax
09341     int    LEVEL0_VECTOR
09342     ret
09343
09344
09345 !***** *-----*
09346 !*          read_tsc      *
09347 !***** *-----*
09348 ! PUBLIC void read_tsc(unsigned long *high, unsigned long *low);
09349 ! Read the cycle counter of the CPU. Pentium and up.
09350 .align 16
09351 _read_tsc:
09352 .data1 0x0f           ! this is the RDTSC instruction
09353 .data1 0x31           ! it places the TSC in EDX:EAX
09354     push   ebp
09355     mov    ebp, 8(esp)
09356     mov    (ebp), edx
09357     mov    ebp, 12(esp)
09358     mov    (ebp), eax
09359     pop    ebp
09360     ret
09361
09362 !***** *-----*
09363 !*          read_flags    *
09364 !***** *-----*
09365 ! PUBLIC unsigned long read_cpu_flags(void);
09366 ! Read CPU status flags from C.
09367 .align 16
09368 _read_cpu_flags:
09369     pushf
09370     mov    eax, (esp)
09371     popf
09372     ret
09373

+++++*-----+
kernel/utility.c
+++++*-----+


09400 /* This file contains a collection of miscellaneous procedures:
09401 *   panic:      abort MINIX due to a fatal error
09402 *   kprintf:    diagnostic output for the kernel
09403 *
09404 * Changes:
09405 *   Dec 10, 2004   kernel printing to circular buffer (Jorrit N. Herder)
09406 *
09407 * This file contains the routines that take care of kernel messages, i.e.,
09408 * diagnostic output within the kernel. Kernel messages are not directly
09409 * displayed on the console, because this must be done by the output driver.
09410 * Instead, the kernel accumulates characters in a buffer and notifies the
09411 * output driver when a new message is ready.
09412 */
09413
09414 #include <minix/com.h>
09415 #include "kernel.h"
09416 #include <stdarg.h>
09417 #include <unistd.h>
09418 #include <stddef.h>
09419 #include <stdlib.h>
09420 #include <signal.h>
09421 #include "proc.h"

```

```

09422 #define END_OF_KMESS -1
09423 FORWARD _PROTOTYPE(void kputc, (int c));
09425
09426 /*=====
09427 *           panic
09428 *=====
09429 PUBLIC void panic(mess,nr)
09430 _CONST char *mess;
09431 int nr;
09432 {
09433 /* The system has run aground of a fatal kernel error. Terminate execution. */
09434     static int panicking = 0;
09435     if (panicking++) return;           /* prevent recursive panics */
09436
09437     if (mess != NULL) {
09438         kprintf("\nKernel panic: %s", mess);
09439         if (nr != NO_NUM) kprintf(" %d", nr);
09440         kprintf("\n",NO_NUM);
09441     }
09442
09443 /* Abort MINIX. */
09444 prepare_shutdown(RBT_PANIC);
09445 }

09447 /*=====
09448 *           kprintf
09449 *=====
09450 PUBLIC void kprintf(const char *fmt, ...)      /* format to be printed */
09451 {
09452     int c;                                /* next character in fmt */
09453     int d;
09454     unsigned long u;                      /* hold number argument */
09455     int base;                            /* base of number arg */
09456     int negative = 0;                     /* print minus sign */
09457     static char x2c[] = "0123456789ABCDEF"; /* nr conversion table */
09458     char ascii[8 * sizeof(long) / 3 + 2]; /* string for ascii number */
09459     char *s = NULL;                       /* string to be printed */
09460     va_list argp;                        /* optional arguments */
09461
09462     va_start(argp, fmt);                /* init variable arguments */
09463
09464     while((c=*fmt++) != 0) {
09465
09466         if (c == '%') {                  /* expect format '%key' */
09467             switch(c = *fmt++) {        /* determine what to do */
09468
09469             /* Known keys are %d, %u, %x, %s, and %%.
09470             * This is easily extended
09471             * with number types like %b and %o by providing a different base.
09472             * Number type keys don't set a string to 's', but use the general
09473             * conversion after the switch statement.
09474
09475             case 'd':                   /* output decimal */
09476                 d = va_arg(argp, signed int);
09477                 if (d < 0) { negative = 1; u = -d; } else { u = d; }
09478                 base = 10;
09479                 break;
09480
09481             case 'u':                   /* output unsigned long */
09482                 u = va_arg(argp, unsigned long);
09483                 base = 10;
09484                 break;
09485
09486             case 'x':                   /* output hexadecimal */
09487                 u = va_arg(argp, unsigned long);
09488                 base = 0x10;
09489                 break;
09490
09491             case 's':                   /* output string */
09492                 s = va_arg(argp, char *);
09493                 if (s == NULL) s = "(null)";
09494
09495         }
09496     }
09497 }

```

```

09490         break;
09491     case '%':                      /* output percent */
09492         s = "%";
09493         break;
09494
09495     /* Unrecognized key. */
09496     default:                         /* echo back %key */
09497         s = "%?";
09498         s[1] = c;                   /* set unknown key */
09499     }
09500
09501     /* Assume a number if no string is set. Convert to ascii. */
09502     if (s == NULL) {
09503         s = ascii + sizeof(ascii)-1;
09504         *s = 0;
09505         do { *--s = x2c((u % base)); } /* work backwards */
09506         while ((u /= base) > 0);
09507     }
09508
09509     /* This is where the actual output for format "%key" is done. */
09510     if (negative) kputc('-');          /* print sign if negative */
09511     while(*s != 0) { kputc(*s++); }    /* print string/ number */
09512     s = NULL;                        /* reset for next round */
09513 }
09514 else {
09515     kputc(c);                      /* print and continue */
09516 }
09517 }
09518 kputc(END_OF_KMESS);                /* terminate output */
09519 va_end(argp);                     /* end variable arguments */
09520 }

09522 /*=====
09523 *           kputc
09524 *=====
09525 PRIVATE void kputc(c)
09526 int c;                            /* character to append */
09527 {
09528 /* Accumulate a single character for a kernel message. Send a notification
09529 * to the output driver if an END_OF_KMESS is encountered.
09530 */
09531 if (c != END_OF_KMESS) {
09532     kmess.km_buf[kmess.km_next] = c; /* put normal char in buffer */
09533     if (kmess.km_size < KMESS_BUF_SIZE)
09534         kmess.km_size += 1;
09535     kmess.km_next = (kmess.km_next + 1) % KMESS_BUF_SIZE;
09536 } else {
09537     send_sig(OUTPUT_PROC_NR, SIGKMESS);
09538 }
09539 }

+++++
kernel/system.h
+++++
09600 /* Function prototypes for the system library.
09601 * The implementation is contained in src/kernel/system/.
09602 *
09603 * The system library allows access to system services by doing a kernel call.
09604 * Kernel calls are transformed into request messages to the SYS task that is
09605 * responsible for handling the call. By convention, sys_call() is transformed
09606 * into a message with type SYS_CALL that is handled in a function do_call().

```

```

09607  */
09608 #ifndef SYSTEM_H
09610 #define SYSTEM_H
09611
09612 /* Common includes for the system library. */
09613 #include "kernel.h"
09614 #include "proto.h"
09615 #include "proc.h"
09616
09617 /* Default handler for unused kernel calls. */
09618 _PROTOTYPE( int do_unused, (message *m_ptr) );
09619 _PROTOTYPE( int do_exec, (message *m_ptr) );
09620 _PROTOTYPE( int do_fork, (message *m_ptr) );
09621 _PROTOTYPE( int do_newmap, (message *m_ptr) );
09622 _PROTOTYPE( int do_exit, (message *m_ptr) );
09623 _PROTOTYPE( int do_trace, (message *m_ptr) );
09624 _PROTOTYPE( int do_nice, (message *m_ptr) );
09625 _PROTOTYPE( int do_copy, (message *m_ptr) );
09626 #define do_vircopy do_copy
09627 #define do_physcopy do_copy
09628 _PROTOTYPE( int do_vcopy, (message *m_ptr) );
09629 #define do_virvcopy do_vcopy
09630 #define do_physvcopy do_vcopy
09631 _PROTOTYPE( int do_umap, (message *m_ptr) );
09632 _PROTOTYPE( int do_memset, (message *m_ptr) );
09633 _PROTOTYPE( int do_abort, (message *m_ptr) );
09634 _PROTOTYPE( int do_getinfo, (message *m_ptr) );
09635 _PROTOTYPE( int do_privctl, (message *m_ptr) );
09636 _PROTOTYPE( int do_segctl, (message *m_ptr) );
09637 _PROTOTYPE( int do_irqctl, (message *m_ptr) );
09638 _PROTOTYPE( int do_devio, (message *m_ptr) );
09639 _PROTOTYPE( int do_vdevio, (message *m_ptr) );
09640 _PROTOTYPE( int do_int86, (message *m_ptr) );
09641 _PROTOTYPE( int do_sdevio, (message *m_ptr) );
09642 _PROTOTYPE( int do_kill, (message *m_ptr) );
09643 _PROTOTYPE( int do_getksig, (message *m_ptr) );
09644 _PROTOTYPE( int do_endksig, (message *m_ptr) );
09645 _PROTOTYPE( int do_sigsend, (message *m_ptr) );
09646 _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
09647 _PROTOTYPE( int do_times, (message *m_ptr) );
09648 _PROTOTYPE( int do_setalarm, (message *m_ptr) );
09649
09650 #endif /* SYSTEM_H */
09651
09652
09653

```

```
+++++
kernel/system.c
+++++
```

```

09700 /* This task provides an interface between the kernel and user-space system
09701 * processes. System services can be accessed by doing a kernel call. Kernel
09702 * calls are transformed into request messages, which are handled by this
09703 * task. By convention, a sys_call() is transformed in a SYS_CALL request
09704 * message that is handled in a function named do_call().
09705 *
09706 * A private call vector is used to map all kernel calls to the functions that
09707 * handle them. The actual handler functions are contained in separate files
09708 * to keep this file clean. The call vector is used in the system task's main
09709 * loop to handle all incoming requests.
09710 *
09711 * In addition to the main sys_task() entry point, which starts the main loop,
09712 * there are several other minor entry points:
09713 *     get_priv:           assign privilege structure to user or system process

```

```

09714 * send_sig:           send a signal directly to a system process
09715 * cause_sig:         take action to cause a signal to occur via PM
09716 * umap_local:        map virtual address in LOCAL_SEG to physical
09717 * umap_remote:       map virtual address in REMOTE_SEG to physical
09718 * umap_bios:         map virtual address in BIOS_SEG to physical
09719 * virtual_copy:      copy bytes from one virtual address to another
09720 * get_randomness:    accumulate randomness in a buffer
09721 *
09722 * Changes:
09723 * Aug 04, 2005   check if kernel call is allowed (Jorrit N. Herder)
09724 * Jul 20, 2005   send signal to services with message (Jorrit N. Herder)
09725 * Jan 15, 2005   new, generalized virtual copy function (Jorrit N. Herder)
09726 * Oct 10, 2004   dispatch system calls from call vector (Jorrit N. Herder)
09727 * Sep 30, 2004   source code documentation updated (Jorrit N. Herder)
09728 */
09729
09730 #include "kernel.h"
09731 #include "system.h"
09732 #include <stdlib.h>
09733 #include <signal.h>
09734 #include <unistd.h>
09735 #include <sys/sigcontext.h>
09736 #include <ibm/memory.h>
09737 #include "protect.h"
09738
09739 /* Declaration of the call vector that defines the mapping of kernel calls
09740 * to handler functions. The vector is initialized in sys_init() with map(),
09741 * which makes sure the kernel call numbers are ok. No space is allocated,
09742 * because the dummy is declared extern. If an illegal call is given, the
09743 * array size will be negative and this won't compile.
09744 */
09745 PUBLIC int (*call_vec[NR_SYS_CALLS])(message *m_ptr);
09746
09747 #define map(call_nr, handler) \
09748     {extern int dummy[NR_SYS_CALLS<(unsigned)(call_nr-KERNEL_CALL) ? 1:-1];} \
09749     call_vec[(call_nr-KERNEL_CALL)] = (handler)
09750
09751 FORWARD _PROTOTYPE( void initialize, (void));
09752
09753 /*=====
09754 *          sys_task
09755 *=====
09756 PUBLIC void sys_task()
09757 {
09758 /* Main entry point of sys_task. Get the message and dispatch on type. */
09759     static message m;
09760     register int result;
09761     register struct proc *caller_ptr;
09762     unsigned int call_nr;
09763     int s;
09764
09765 /* Initialize the system task. */
09766     initialize();
09767
09768     while (TRUE) {
09769         /* Get work. Block and wait until a request message arrives. */
09770         receive(ANY, &m);
09771         call_nr = (unsigned) m.m_type - KERNEL_CALL;
09772         caller_ptr = proc_addr(m.m_source);
09773
09774         /* See if the caller made a valid request and try to handle it. */
09775         if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
09776             kprintf("SYSTEM: request %d from %d denied.\n", call_nr,m.m_source);
09777             result = ECALLDENIED;           /* illegal message type */
09778         } else if (call_nr >= NR_SYS_CALLS) {           /* check call number */
09779             kprintf("SYSTEM: illegal request %d from %d.\n", call_nr,m.m_source);
09780             result = EBADREQUEST;          /* illegal message type */

```

```

09781      }
09782  else {
09783      result = (*call_vec[call_nr])(&m); /* handle the kernel call */
09784  }
09785
09786  /* Send a reply, unless inhibited by a handler function. Use the kernel
09787  * function lock_send() to prevent a system call trap. The destination
09788  * is known to be blocked waiting for a message.
09789  */
09790  if (result != EDONTREPLY) {
09791      m.m_type = result;           /* report status of call */
09792      if (OK != (s=lock_send(m.m_source, &m))) {
09793          kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
09794      }
09795  }
09796 }
09797 }

09799 /*=====
09800 *           initialize
09801 *=====
09802 PRIVATE void initialize(void)
09803 {
09804     register struct priv *sp;
09805     int i;
09806
09807     /* Initialize IRQ handler hooks. Mark all hooks available. */
09808     for (i=0; i<NR_IRQ_HOOKS; i++) {
09809         irq_hooks[i].proc_nr = NONE;
09810     }
09811
09812     /* Initialize all alarm timers for all processes. */
09813     for (sp=BEG_PRIV_ADDR; sp < END_PRIV_ADDR; sp++) {
09814         tmr_inittimer(&(sp->s_alarm_timer));
09815     }
09816
09817     /* Initialize the call vector to a safe default handler. Some kernel calls
09818     * may be disabled or nonexistent. Then explicitly map known calls to their
09819     * handler functions. This is done with a macro that gives a compile error
09820     * if an illegal call number is used. The ordering is not important here.
09821     */
09822     for (i=0; i<NR_SYS_CALLS; i++) {
09823         call_vec[i] = do_unused;
09824     }
09825
09826     /* Process management. */
09827     map(SYS_FORK, do_fork);           /* a process forked a new process */
09828     map(SYS_EXEC, do_exec);          /* update process after execute */
09829     map(SYS_EXIT, do_exit);          /* clean up after process exit */
09830     map(SYS_NICE, do_nice);          /* set scheduling priority */
09831     map(SYS_PRIVCTL, do_privctl);   /* system privileges control */
09832     map(SYS_TRACE, do_trace);        /* request a trace operation */
09833
09834     /* Signal handling. */
09835     map(SYS_KILL, do_kill);          /* cause a process to be signaled */
09836     map(SYS_GETKSIG, do_getksig);    /* PM checks for pending signals */
09837     map(SYS_ENDKSIG, do_endksig);    /* PM finished processing signal */
09838     map(SYS_SIGSEND, do_sigsend);    /* start POSIX-style signal */
09839     map(SYS_SIGRETURN, do_sigreturn); /* return from POSIX-style signal */
09840
09841     /* Device I/O. */
09842     map(SYS_IRQCTL, do_irqctl);     /* interrupt control operations */
09843     map(SYS_DEVIO, do_devio);        /* inb, inw, inl, outb, outw, outl */
09844     map(SYS_SDEVIO, do_sdevio);      /* phys_insb, _insw, _outsb, _outsw */

```

```

09845 map(SYS_VDEVIO, do_vdevio);           /* vector with devio requests */
09846 map(SYS_INT86, do_int86);           /* real-mode BIOS calls */
09847
09848 /* Memory management. */
09849 map(SYS_NEWMAP, do_newmap);          /* set up a process memory map */
09850 map(SYS_SEGCTL, do_segctl);          /* add segment and get selector */
09851 map(SYS_MEMSET, do_memset);          /* write char to memory area */
09852
09853 /* Copying. */
09854 map(SYS_UMAP, do_umap);             /* map virtual to physical address */
09855 map(SYS_VIRCOPY, do_vircopy);        /* use pure virtual addressing */
09856 map(SYS_PHYSCOPY, do_physcopy);      /* use physical addressing */
09857 map(SYS_VIRVCOPY, do_virvcopy);       /* vector with copy requests */
09858 map(SYS_PHYSVCOPY, do_physvcopy);     /* vector with copy requests */
09859
09860 /* Clock functionality. */
09861 map(SYS_TIMES, do_times);           /* get uptime and process times */
09862 map(SYS_SETALARM, do_setalarm);      /* schedule a synchronous alarm */
09863
09864 /* System control. */
09865 map(SYS_ABORT, do_abort);            /* abort MINIX */
09866 map(SYS_GETINFO, do_getinfo);         /* request system information */
09867 }

09869 /*=====
09870 *          get_priv
09871 *=====
09872 PUBLIC int get_priv(rc, proc_type)
09873 register struct proc *rc;           /* new (child) process pointer */
09874 int proc_type;                     /* system or user process flag */
09875 {
09876 /* Get a privilege structure. All user processes share the same privilege
09877 * structure. System processes get their own privilege structure.
09878 */
09879 register struct priv *sp;           /* privilege structure */
09880
09881 if (proc_type == SYS_PROC) {          /* find a new slot */
09882     for (sp = BEG_PRIV_ADDR; sp < END_PRIV_ADDR; ++sp)
09883         if (sp->s_proc_nr == NONE && sp->s_id != USER_PRIV_ID) break;
09884     if (sp->s_proc_nr != NONE) return(ENOSPC);
09885     rc->p_priv = sp;                  /* assign new slot */
09886     rc->p_priv->s_proc_nr = proc_nr(rc); /* set association */
09887     rc->p_priv->s_flags = SYS_PROC;   /* mark as privileged */
09888 } else {
09889     rc->p_priv = &priv[USER_PRIV_ID]; /* use shared slot */
09890     rc->p_priv->s_proc_nr = INIT_PROC_NR; /* set association */
09891     rc->p_priv->s_flags = 0;          /* no initial flags */
09892 }
09893 return(OK);
09894 }

09896 /*=====
09897 *          get_randomness
09898 *=====
09899 PUBLIC void get_randomness(source)
09900 int source;
09901 {
09902 /* On machines with the RDTSC (cycle counter read instruction - pentium
09903 * and up), use that for high-resolution raw entropy gathering. Otherwise,
09904 * use the realtime clock (tick resolution).
09905 */
09906 /* Unfortunately this test is run-time - we don't want to bother with
09907 * compiling different kernels for different machines.
09908 */
09909 /* On machines without RDTSC, we use read_clock().
09910 */

```

```

09911     int r_next;
09912     unsigned long tsc_high, tsc_low;
09913
09914     source %= RANDOM_SOURCES;
09915     r_next= krandom.bin[source].r_next;
09916     if (machine.processor > 486) {
09917         read_tsc(&tsc_high, &tsc_low);
09918         krandom.bin[source].r_buf[r_next] = tsc_low;
09919     } else {
09920         krandom.bin[source].r_buf[r_next] = read_clock();
09921     }
09922     if (krandom.bin[source].r_size < RANDOM_ELEMENTS) {
09923         krandom.bin[source].r_size++;
09924     }
09925     krandom.bin[source].r_next = (r_next + 1) % RANDOM_ELEMENTS;
09926 }

09928 /*=====
09929 *          send_sig
09930 *=====
09931 PUBLIC void send_sig(proc_nr, sig_nr)
09932 int proc_nr;           /* system process to be signalled */
09933 int sig_nr;            /* signal to be sent, 1 to _NSIG */
09934 {
09935 /* Notify a system process about a signal. This is straightforward. Simply
09936 * set the signal that is to be delivered in the pending signals map and
09937 * send a notification with source SYSTEM.
09938 */
09939     register struct proc *rp;
09940
09941     rp = proc_addr(proc_nr);
09942     sigaddset(&priv(rp)->s_sig_pending, sig_nr);
09943     lock_notify(SYSTEM, proc_nr);
09944 }

09946 /*=====
09947 *          cause_sig
09948 *=====
09949 PUBLIC void cause_sig(proc_nr, sig_nr)
09950 int proc_nr;           /* process to be signalled */
09951 int sig_nr;            /* signal to be sent, 1 to _NSIG */
09952 {
09953 /* A system process wants to send a signal to a process. Examples are:
09954 * - HARDWARE wanting to cause a SIGSEGV after a CPU exception
09955 * - TTY wanting to cause SIGINT upon getting a DEL
09956 * - FS wanting to cause SIGPIPE for a broken pipe
09957 * Signals are handled by sending a message to PM. This function handles the
09958 * signals and makes sure the PM gets them by sending a notification. The
09959 * process being signaled is blocked while PM has not finished all signals
09960 * for it.
09961 * Race conditions between calls to this function and the system calls that
09962 * process pending kernel signals cannot exist. Signal related functions are
09963 * only called when a user process causes a CPU exception and from the kernel
09964 * process level, which runs to completion.
09965 */
09966     register struct proc *rp;
09967
09968 /* Check if the signal is already pending. Process it otherwise. */
09969     rp = proc_addr(proc_nr);
09970     if (!sigismember(&rp->p_pending, sig_nr)) {
09971         sigaddset(&rp->p_pending, sig_nr);
09972         if (!(rp->p_rts_flags & SIGNALED)) {           /* other pending */
09973             if (rp->p_rts_flags == 0) lock_dequeue(rp); /* make not ready */
09974             rp->p_rts_flags |= SIGNALED | SIC_PENDING; /* update flags */
09975             send_sig(PM_PROC_NR, SIGKSIG);
09976         }
09977     }

```

```

09978 }

09980 /*=====
09981 *                               umap_local
09982 *=====*/
09983 PUBLIC phys_bytes umap_local(rp, seg, vir_addr, bytes)
09984 register struct proc *rp;           /* pointer to proc table entry for process */
09985 int seg;                          /* T, D, or S segment */
09986 vir_bytes vir_addr;              /* virtual address in bytes within the seg */
09987 vir_bytes bytes;                 /* # of bytes to be copied */
09988 {
09989 /* Calculate the physical memory address for a given virtual address. */
09990     vir_ticks vc;                  /* the virtual address in ticks */
09991     phys_bytes pa;                /* intermediate variables as phys_bytes */
09992     phys_bytes seg_base;
09993
09994 /* If 'seg' is D it could really be S and vice-versa. T really means T.
09995 * If the virtual address falls in the gap, it causes a problem. On the
09996 * 8088 it is probably a legal stack reference, since "stackfaults" are
09997 * not detected by the hardware. On 8088s, the gap is called S and
09998 * accepted, but on other machines it is called D and rejected.
09999 * The Atari ST behaves like the 8088 in this respect.
10000 */
10001
10002 if (bytes <= 0) return( (phys_bytes) 0);
10003 if (vir_addr + bytes <= vir_addr) return 0; /* overflow */
10004 vc = (vir_addr + bytes - 1) >> CLICK_SHIFT; /* last click of data */
10005
10006 if (seg != T)
10007     seg = (vc < rp->p_memmap[0].mem_vir + rp->p_memmap[D].mem_len ? D : S);
10008
10009 if ((vir_addr>>CLICK_SHIFT) >= rp->p_memmap[seg].mem_vir +
10010     rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );
10011
10012 if (vc >= rp->p_memmap[seg].mem_vir +
10013     rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );
10014
10015 seg_base = (phys_bytes) rp->p_memmap[seg].mem_phys;
10016 seg_base = seg_base << CLICK_SHIFT; /* segment origin in bytes */
10017 pa = (phys_bytes) vir_addr;
10018 pa -= rp->p_memmap[seg].mem_vir << CLICK_SHIFT;
10019 return(seg_base + pa);
10020 }

10022 /*=====
10023 *                               umap_remote
10024 *=====*/
10025 PUBLIC phys_bytes umap_remote(rp, seg, vir_addr, bytes)
10026 register struct proc *rp;           /* pointer to proc table entry for process */
10027 int seg;                          /* index of remote segment */
10028 vir_bytes vir_addr;              /* virtual address in bytes within the seg */
10029 vir_bytes bytes;                 /* # of bytes to be copied */
10030 {
10031 /* Calculate the physical memory address for a given virtual address. */
10032     struct far_mem *fm;
10033
10034 if (bytes <= 0) return( (phys_bytes) 0);
10035 if (seg < 0 || seg >= NR_REMOTE_SEGS) return( (phys_bytes) 0);
10036
10037 fm = &rp->p_priv->s_farmem[seg];
10038 if (! fm->in_use) return( (phys_bytes) 0);
10039 if (vir_addr + bytes > fm->mem_len) return( (phys_bytes) 0);
10040
10041 return(fm->mem_phys + (phys_bytes) vir_addr);
10042 }

```

```

10044 /*=====
10045 *          umap_bios
10046 *=====
10047 PUBLIC phys_bytes umap_bios(rp, vir_addr, bytes)
10048 register struct proc *rp;           /* pointer to proc table entry for process */
10049 vir_bytes vir_addr;               /* virtual address in BIOS segment */
10050 vir_bytes bytes;                /* # of bytes to be copied */
10051 {
10052 /* Calculate the physical memory address at the BIOS. Note: currently, BIOS
10053 * address zero (the first BIOS interrupt vector) is not considered as an
10054 * error here, but since the physical address will be zero as well, the
10055 * calling function will think an error occurred. This is not a problem,
10056 * since no one uses the first BIOS interrupt vector.
10057 */
10058
10059 /* Check all acceptable ranges. */
10060 if (vir_addr >= BIOS_MEM_BEGIN && vir_addr + bytes <= BIOS_MEM_END)
10061     return (phys_bytes) vir_addr;
10062 else if (vir_addr >= BASE_MEM_TOP && vir_addr + bytes <= UPPER_MEM_END)
10063     return (phys_bytes) vir_addr;
10064 kprintf("Warning, error in umap_bios, virtual address 0x%lx\n", vir_addr);
10065 return 0;
10066 }

10068 /*=====
10069 *          virtual_copy
10070 *=====
10071 PUBLIC int virtual_copy(src_addr, dst_addr, bytes)
10072 struct vir_addr *src_addr;          /* source virtual address */
10073 struct vir_addr *dst_addr;          /* destination virtual address */
10074 vir_bytes bytes;                 /* # of bytes to copy */
10075 {
10076 /* Copy bytes from virtual address src_addr to virtual address dst_addr.
10077 * Virtual addresses can be in ABS, LOCAL_SEG, REMOTE_SEG, or BIOS_SEG.
10078 */
10079 struct vir_addr *vir_addr[2]; /* virtual source and destination address */
10080 phys_bytes phys_addr[2];        /* absolute source and destination */
10081 int seg_index;
10082 int i;
10083
10084 /* Check copy count. */
10085 if (bytes <= 0) return(EDOM);
10086
10087 /* Do some more checks and map virtual addresses to physical addresses. */
10088 vir_addr[_SRC_] = src_addr;
10089 vir_addr[_DST_] = dst_addr;
10090 for (i=_SRC_; i<=_DST_; i++) {
10091
10092 /* Get physical address. */
10093 switch((vir_addr[i]->segment & SEGMENT_TYPE)) {
10094 case LOCAL_SEG:
10095     seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
10096     phys_addr[i] = umap_local( proc_addr(vir_addr[i]->proc_nr),
10097         seg_index, vir_addr[i]->offset, bytes );
10098     break;
10099 case REMOTE_SEG:
10100     seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
10101     phys_addr[i] = umap_remote( proc_addr(vir_addr[i]->proc_nr),
10102         seg_index, vir_addr[i]->offset, bytes );
10103     break;
10104 case BIOS_SEG:
10105     phys_addr[i] = umap_bios( proc_addr(vir_addr[i]->proc_nr),
10106         vir_addr[i]->offset, bytes );
10107     break;
10108 case PHYS_SEG:
10109     phys_addr[i] = vir_addr[i]->offset;
10110     break;

```

```

10111     default:
10112         return(EINVAL);
10113     }
10114
10115     /* Check if mapping succeeded. */
10116     if (phys_addr[i] <= 0 && vir_addr[i]->segment != PHYS_SEG)
10117         return(EFAULT);
10118     }
10119
10120     /* Now copy bytes between physical addresseses. */
10121     phys_copy(phys_addr[_SRC_], phys_addr[_DST_], (phys_bytes) bytes);
10122     return(OK);
10123 }
```

```

+++++
kernel/system/do_setalarm.c
+++++
10200 /* The kernel call implemented in this file:
10201 *   m_type:    SYS_SETALARM
10202 *
10203 * The parameters for this kernel call are:
10204 *   m2_i1:    ALRM_EXP_TIME          (alarm's expiration time)
10205 *   m2_i2:    ALRM_ABS_TIME         (expiration time is absolute?)
10206 *   m2_i1:    ALRM_TIME_LEFT        (return seconds left of previous)
10207 */
10208
10209 #include "../system.h"
10210
10211 #if USE_SETALARM
10212
10213 FORWARD _PROTOTYPE( void cause_alarm, (timer_t *tp) );
10214
10215 /*=====
10216 *           do_setalarm
10217 *=====
10218 PUBLIC int do_setalarm(m_ptr)
10219 message *m_ptr;           /* pointer to request message */
10220 {
10221 /* A process requests a synchronous alarm, or wants to cancel its alarm. */
10222 register struct proc *rp;    /* pointer to requesting process */
10223 int proc_nr;              /* which process wants the alarm */
10224 long exp_time;            /* expiration time for this alarm */
10225 int use_abs_time;          /* use absolute or relative time */
10226 timer_t *tp;              /* the process' timer structure */
10227 clock_t uptime;           /* placeholder for current uptime */
10228
10229 /* Extract shared parameters from the request message. */
10230 exp_time = m_ptr->ALRM_EXP_TIME;      /* alarm's expiration time */
10231 use_abs_time = m_ptr->ALRM_ABS_TIME;  /* flag for absolute time */
10232 proc_nr = m_ptr->m_source;            /* process to interrupt later */
10233 rp = proc_addr(proc_nr);             /* */
10234 if (! (priv(rp)->s_flags & SYS_PROC)) return(EPERM);
10235
10236 /* Get the timer structure and set the parameters for this alarm. */
10237 tp = &(priv(rp)->s_alarm_timer);
10238 tmr_arg(tp)->ta_int = proc_nr;
10239 tp->tmr_func = cause_alarm;
10240
10241 /* Return the ticks left on the previous alarm. */
10242 uptime = get_uptime();
10243 if ((tp->tmr_exp_time != TMR_NEVER) && (uptime < tp->tmr_exp_time) ) {
10244     m_ptr->ALRM_TIME_LEFT = (tp->tmr_exp_time - uptime);
10245 } else {
```

```

10246     m_ptr->ALRM_TIME_LEFT = 0;
10247 }
10248
10249 /* Finally, (re)set the timer depending on the expiration time. */
10250 if (exp_time == 0) {
10251     reset_timer(tp);
10252 } else {
10253     tp->tmr_exp_time = (use_abs_time) ? exp_time : exp_time + get_uptime();
10254     set_timer(tp, tp->tmr_exp_time, tp->tmr_func);
10255 }
10256 return(OK);
10257 }

10259 /*=====
10260 *          cause_alarm
10261 *=====
10262 PRIVATE void cause_alarm(tp)
10263 timer_t *tp;
10264 {
10265 /* Routine called if a timer goes off and the process requested a synchronous
10266 * alarm. The process number is stored in timer argument 'ta_int'. Notify that
10267 * process with a notification message from CLOCK.
10268 */
10269     int proc_nr = tmr_arg(tp)->ta_int;           /* get process number */
10270     lock_notify(CLOCK, proc_nr);                  /* notify process */
10271 }

10273 #endif /* USE_SETALARM */

```

+++++  
kernel/system/do\_exec.c  
+++++

```

10300 /* The kernel call implemented in this file:
10301 *   m_type:    SYS_EXEC
10302 *
10303 * The parameters for this kernel call are:
10304 *   m1_i1:    PR_PROC_NR           (process that did exec call)
10305 *   m1_p1:    PR_STACK_PTR        (new stack pointer)
10306 *   m1_p2:    PR_NAME_PTR         (pointer to program name)
10307 *   m1_p3:    PR_IP_PTR          (new instruction pointer)
10308 */
10309 #include "../system.h"
10310 #include <string.h>
10311 #include <signal.h>
10312
10313 #if USE_EXEC
10314
10315 /*=====
10316 *          do_exec
10317 *=====
10318 PUBLIC int do_exec(m_ptr)
10319 register message *m_ptr;      /* pointer to request message */
10320 {
10321 /* Handle sys_exec(). A process has done a successful EXEC. Patch it up. */
10322     register struct proc *rp;
10323     reg_t sp;                      /* new sp */
10324     phys_bytes phys_name;
10325     char *np;
10326
10327     rp = proc_addr(m_ptr->PR_PROC_NR);
10328     sp = (reg_t) m_ptr->PR_STACK_PTR;
10329     rp->p_reg.sp = sp;             /* set the stack pointer */
10330     phys_memset(vir2phys(&rp->p_ldt[EXTRA_LDT_INDEX]), 0,
10331                 (LDT_SIZE - EXTRA_LDT_INDEX) * sizeof(rp->p_ldt[0]));
10332     rp->p_reg.pc = (reg_t) m_ptr->PR_IP_PTR;           /* set pc */
10333     rp->p_rts_flags &= ~RECEIVING;           /* PM does not reply to EXEC call */

```

```

10334     if (rp->p_rts_flags == 0) lock_enqueue(rp);
10335
10336     /* Save command name for debugging, ps(1) output, etc. */
10337     phys_name = numap_local(m_ptr->m_source, (vir_bytes) m_ptr->PR_NAME_PTR,
10338                               (vir_bytes) P_NAME_LEN - 1);
10339     if (phys_name != 0) {
10340         phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) P_NAME_LEN - 1);
10341         for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
10342         *np = 0;                                /* mark end */
10343     } else {
10344         strncpy(rp->p_name, "<unset>", P_NAME_LEN);
10345     }
10346     return(OK);
10347 }
10348 #endif /* USE_EXEC */

+++++
kernel/clock.c
+++++

10400 /* This file contains the clock task, which handles time related functions.
10401 * Important events that are handled by the CLOCK include setting and
10402 * monitoring alarm timers and deciding when to (re)schedule processes.
10403 * The CLOCK offers a direct interface to kernel processes. System services
10404 * can access its services through system calls, such as sys_setalarm(). The
10405 * CLOCK task thus is hidden from the outside world.
10406 *
10407 * Changes:
10408 *   Oct 08, 2005  reordering and comment editing (A. S. Woodhull)
10409 *   Mar 18, 2004  clock interface moved to SYSTEM task (Jorrit N. Herder)
10410 *   Sep 30, 2004  source code documentation updated (Jorrit N. Herder)
10411 *   Sep 24, 2004  redesigned alarm timers (Jorrit N. Herder)
10412 *
10413 * The function do_clocktick() is triggered by the clock's interrupt
10414 * handler when a watchdog timer has expired or a process must be scheduled.
10415 *
10416 * In addition to the main clock_task() entry point, which starts the main
10417 * loop, there are several other minor entry points:
10418 *   clock_stop:      called just before MINIX shutdown
10419 *   get_uptime:      get realtime since boot in clock ticks
10420 *   set_timer:       set a watchdog timer (+)
10421 *   reset_timer:    reset a watchdog timer (+)
10422 *   read_clock:     read the counter of channel 0 of the 8253A timer
10423 *
10424 * (+) The CLOCK task keeps tracks of watchdog timers for the entire kernel.
10425 * The watchdog functions of expired timers are executed in do_clocktick().
10426 * It is crucial that watchdog functions not block, or the CLOCK task may
10427 * be blocked. Do not send() a message when the receiver is not expecting it.
10428 * Instead, notify(), which always returns, should be used.
10429 */
10430
10431 #include "kernel.h"
10432 #include "proc.h"
10433 #include <signal.h>
10434 #include <minix/com.h>
10435
10436 /* Function prototype for PRIVATE functions. */
10437 FORWARD _PROTOTYPE( void init_clock, (void) );
10438 FORWARD _PROTOTYPE( int clock_handler, (irq_hook_t *hook) );
10439 FORWARD _PROTOTYPE( int do_clocktick, (message *m_ptr) );
10440
10441 /* Clock parameters. */
10442 #define COUNTER_FREQ (2*TIMER_FREQ) /* counter frequency using square wave */
10443 #define LATCH_COUNT 0x00 /* cc0xxxx, c = channel, x = any */
10444 #define SQUARE_WAVE 0x36 /* ccaammb, a = access, m = mode, b = BCD */
10445             /* 11x11, 11 = LSB then MSB, x11 = sq wave */

```

```

10446 #define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
10447 #define TIMER_FREQ 1193182L /* clock frequency for timer in PC and AT */
10448
10449 #define CLOCK_ACK_BIT 0x80 /* PS/2 clock interrupt acknowledge bit */
10450
10451 /* The CLOCK's timers queue. The functions in <timers.h> operate on this.
10452 * Each system process possesses a single synchronous alarm timer. If other
10453 * kernel parts want to use additional timers, they must declare their own
10454 * persistent (static) timer structure, which can be passed to the clock
10455 * via (re)set_timer().
10456 * When a timer expires its watchdog function is run by the CLOCK task.
10457 */
10458 PRIVATE timer_t *clock_timers; /* queue of CLOCK timers */
10459 PRIVATE clock_t next_timeout; /* realtime that next timer expires */
10460
10461 /* The time is incremented by the interrupt handler on each clock tick. */
10462 PRIVATE clock_t realtime; /* real time clock */
10463 PRIVATE irq_hook_t clock_hook; /* interrupt handler hook */
10464
10465 /*=====
10466 *                               clock_task
10467 *=====
10468 PUBLIC void clock_task()
10469 {
10470 /* Main program of clock task. If the call is not HARD_INT it is an error.
10471 */
10472     message m; /* message buffer for both input and output */
10473     int result; /* result returned by the handler */
10474
10475     init_clock(); /* initialize clock task */
10476
10477 /* Main loop of the clock task. Get work, process it. Never reply. */
10478 while (TRUE) {
10479
10480     /* Go get a message. */
10481     receive(ANY, &m);
10482
10483     /* Handle the request. Only clock ticks are expected. */
10484     switch (m.m_type) {
10485         case HARD_INT:
10486             result = do_clocktick(&m); /* handle clock tick */
10487             break;
10488         default: /* illegal request type */
10489             kprintf("CLOCK: illegal request %d from %d.\n", m.m_type, m.m_source);
10490     }
10491 }
10492 }

10493 /*=====
10494 *                               do_clocktick
10495 *=====
10496 PRIVATE int do_clocktick(m_ptr)
10497 message *m_ptr; /* pointer to request message */
10498 {
10499 /* Despite its name, this routine is not called on every clock tick. It
10500 * is called on those clock ticks when a lot of work needs to be done.
10501 */
10502
10503 /* A process used up a full quantum. The interrupt handler stored this
10504 * process in 'prev_ptr'. First make sure that the process is not on the
10505 * scheduling queues. Then announce the process ready again. Since it has
10506 * no more time left, it gets a new quantum and is inserted at the right
10507 * place in the queues. As a side-effect a new process will be scheduled.
10508 */
10509 if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEMPTIBLE) {
10510     lock_dequeue(prev_ptr); /* take it off the queues */
10511     lock_enqueue(prev_ptr); /* and reinsert it again */
10512 }
```

```

10513     }
10514
10515     /* Check if a clock timer expired and run its watchdog function. */
10516     if (next_timeout <= realtime) {
10517         tmrs_exptimers(&clock_timers, realtime, NULL);
10518         next_timeout = clock_timers == NULL ?
10519                         TMR_NEVER : clock_timers->tmr_exp_time;
10520     }
10521
10522     /* Inhibit sending a reply. */
10523     return(EDONTREPLY);
10524 }

10526 /*=====
10527 *          init_clock
10528 *=====
10529 PRIVATE void init_clock()
10530 {
10531     /* Initialize the CLOCK's interrupt hook. */
10532     clock_hook.proc_nr = CLOCK;
10533
10534     /* Initialize channel 0 of the 8253A timer to, e.g., 60 Hz. */
10535     outb(TIMER_MODE, SQUARE_WAVE);           /* set timer to run continuously */
10536     outb(TIMERO, TIMER_COUNT);             /* load timer low byte */
10537     outb(TIMERO, TIMER_COUNT >> 8);        /* load timer high byte */
10538     put_irq_handler(&clock_hook, CLOCK_IRQ, clock_handler);/* register handler */
10539     enable_irq(&clock_hook);               /* ready for clock interrupts */
10540 }

10542 /*=====
10543 *          clock_stop
10544 *=====
10545 PUBLIC void clock_stop()
10546 {
10547     /* Reset the clock to the BIOS rate. (For rebooting) */
10548     outb(TIMER_MODE, 0x36);
10549     outb(TIMERO, 0);
10550     outb(TIMERO, 0);
10551 }

10553 /*=====
10554 *          clock_handler
10555 *=====
10556 PRIVATE int clock_handler(hook)
10557 irq_hook_t *hook;
10558 {
10559     /* This executes on each clock tick (i.e., every time the timer chip generates
10560     * an interrupt). It does a little bit of work so the clock task does not have
10561     * to be called on every tick. The clock task is called when:
10562     *
10563     *      (1) the scheduling quantum of the running process has expired, or
10564     *      (2) a timer has expired and the watchdog function should be run.
10565     *
10566     * Many global global and static variables are accessed here. The safety of
10567     * this must be justified. All scheduling and message passing code acquires a
10568     * lock by temporarily disabling interrupts, so no conflicts with calls from
10569     * the task level can occur. Furthermore, interrupts are not reentrant, the
10570     * interrupt handler cannot be bothered by other interrupts.
10571     *
10572     * Variables that are updated in the clock's interrupt handler:
10573     *     lost_ticks:
10574     *         Clock ticks counted outside the clock task. This for example
10575     *         is used when the boot monitor processes a real mode interrupt.
10576     *     realtime:
10577     *         The current uptime is incremented with all outstanding ticks.
10578     *     proc_ptr, bill_ptr:
10579     *         These are used for accounting. It does not matter if proc.c

```

```

10580 *           is changing them, provided they are always valid pointers,
10581 *           since at worst the previous process would be billed.
10582 */
10583 register unsigned ticks;
10584
10585 /* Acknowledge the PS/2 clock interrupt. */
10586 if (machine.ps_mca) outb(PORT_B, inb(PORT_B) | CLOCK_ACK_BIT);
10587
10588 /* Get number of ticks and update realtime. */
10589 ticks = lost_ticks + 1;
10590 lost_ticks = 0;
10591 realtime += ticks;
10592
10593 /* Update user and system accounting times. Charge the current process for
10594 * user time. If the current process is not billable, that is, if a non-user
10595 * process is running, charge the billable process for system time as well.
10596 * Thus the unbillable process' user time is the billable user's system time.
10597 */
10598 proc_ptr->p_user_time += ticks;
10599 if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
10600     proc_ptr->p_ticks_left -= ticks;
10601 }
10602 if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
10603     bill_ptr->p_sys_time += ticks;
10604     bill_ptr->p_ticks_left -= ticks;
10605 }
10606
10607 /* Check if do_clocktick() must be called. Done for alarms and scheduling.
10608 * Some processes, such as the kernel tasks, cannot be preempted.
10609 */
10610 if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0)) {
10611     prev_ptr = proc_ptr;                                /* store running process */
10612     lock_notify(HARDWARE, CLOCK);                      /* send notification */
10613 }
10614 return(1);                                         /* reenable interrupts */
10615 }

10616 /*=====
10617 *          get_uptime
10618 *=====
10619 PUBLIC clock_t get_uptime()
10620 {
10621 /* Get and return the current clock uptime in ticks. */
10622     return(realtime);
10623 }
10624 */

10625 /*=====
10626 *          set_timer
10627 *=====
10628 PUBLIC void set_timer(tp, exp_time, watchdog)
10629 struct timer *tp;                                     /* pointer to timer structure */
10630 clock_t exp_time;                                    /* expiration realtime */
10631 tmr_func_t watchdog;                               /* watchdog to be called */
10632 {
10633 /* Insert the new timer in the active timers list. Always update the
10634 * next timeout time by setting it to the front of the active list.
10635 */
10636     tmrs_settimer(&clock_timers, tp, exp_time, watchdog, NULL);
10637     next_timeout = clock_timers->tmr_exp_time;
10638 }
10639 */

10640 /*=====
10641 *          reset_timer
10642 *=====
10643 */
10644 PUBLIC void reset_timer(tp)
10645 struct timer *tp;                                     /* pointer to timer structure */
10646 {

```

```

10647 /* The timer pointed to by 'tp' is no longer needed. Remove it from both the
10648 * active and expired lists. Always update the next timeout time by setting
10649 * it to the front of the active list.
10650 */
10651 tmrs_clrtimer(&clock_timers, tp, NULL);
10652 next_timeout = (clock_timers == NULL) ?
10653     TMR_NEVER : clock_timers->tmr_exp_time;
10654 }

10655 /*=====
10656 *          read_clock
10657 *=====
10658 */
10659 PUBLIC unsigned long read_clock()
10660 {
10661 /* Read the counter of channel 0 of the 8253A timer. This counter counts
10662 * down at a rate of TIMER_FREQ and restarts at TIMER_COUNT-1 when it
10663 * reaches zero. A hardware interrupt (clock tick) occurs when the counter
10664 * gets to zero and restarts its cycle.
10665 */
10666     unsigned count;
10667
10668     outb(TIMER_MODE, LATCH_COUNT);
10669     count = inb(TIMERO);
10670     count |= (inb(TIMERO) << 8);
10671
10672     return count;
10673 }

```

```
=====
drivers/drivers.h
=====
```

```

10700 /* This is the master header for all device drivers. It includes some other
10701 * files and defines the principal constants.
10702 */
10703 #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
10704 #define _MINIX             1    /* tell headers to include MINIX stuff */
10705 #define _SYSTEM            1    /* get negative error number in <errno.h> */
10706
10707 /* The following are so basic, all the *.c files get them automatically. */
10708 #include <minix/config.h>        /* MUST be first */
10709 #include <ansi.h>                /* MUST be second */
10710 #include <minix/type.h>
10711 #include <minix/com.h>
10712 #include <minix/dmap.h>
10713 #include <minix/callnr.h>
10714 #include <sys/types.h>
10715 #include <minix/const.h>
10716 #include <minix/devio.h>
10717 #include <minix/syslib.h>
10718 #include <minix/sysutil.h>
10719 #include <minix/bitmap.h>
10720
10721 #include <ibm/interrupt.h>        /* IRQ vectors and miscellaneous ports */
10722 #include <ibm/bios.h>              /* BIOS index numbers */
10723 #include <ibm/ports.h>             /* Well-known ports */
10724
10725 #include <string.h>
10726 #include <signal.h>
10727 #include <stdlib.h>
10728 #include <limits.h>
10729 #include <stddef.h>
10730 #include <errno.h>
10731 #include <unistd.h>
10732

```

```
+++++
drivers/libdriver/driver.h
+++++

10800 /* Types and constants shared between the generic and device dependent
10801   * device driver code.
10802   */
10803
10804 #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
10805 #define _MINIX             1    /* tell headers to include MINIX stuff */
10806 #define _SYSTEM            1    /* get negative error number in <errno.h> */
10807
10808 /* The following are so basic, all the *.c files get them automatically. */
10809 #include <minix/config.h>      /* MUST be first */
10810 #include <ansi.h>           /* MUST be second */
10811 #include <minix/type.h>
10812 #include <minix/ ipc.h>
10813 #include <minix/com.h>
10814 #include <minix/callnr.h>
10815 #include <sys/types.h>
10816 #include <minix/const.h>
10817 #include <minix/syslib.h>
10818 #include <minix/sysutil.h>
10819
10820 #include <string.h>
10821 #include <limits.h>
10822 #include <stddef.h>
10823 #include <errno.h>
10824
10825 #include <minix/partition.h>
10826 #include <minix/u64.h>
10827
10828 /* Info about and entry points into the device dependent code. */
10829 struct driver {
10830     _PROTOTYPE( char (*dr_name), (void) );
10831     _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
10832     _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
10833     _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
10834     _PROTOTYPE( struct device *(*dr_prepare), (int device) );
10835     _PROTOTYPE( int (*dr_transfer), (int proc_nr, int opcode, off_t position,
10836                                         iovec_t *iov, unsigned nr_req) );
10837     _PROTOTYPE( void (*dr_cleanup), (void) );
10838     _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
10839     _PROTOTYPE( void (*dr_signal), (struct driver *dp, message *m_ptr) );
10840     _PROTOTYPE( void (*dr_alarm), (struct driver *dp, message *m_ptr) );
10841     _PROTOTYPE( int (*dr_cancel), (struct driver *dp, message *m_ptr) );
10842     _PROTOTYPE( int (*dr_select), (struct driver *dp, message *m_ptr) );
10843     _PROTOTYPE( int (*dr_other), (struct driver *dp, message *m_ptr) );
10844     _PROTOTYPE( int (*dr_hw_int), (struct driver *dp, message *m_ptr) );
10845 };
10846
10847 #if (CHIP == INTEL)
10848
10849 /* Number of bytes you can DMA before hitting a 64K boundary: */
10850 #define dma_bytes_left(phys) \
10851     ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))
10852
10853 #endif /* CHIP == INTEL */
10854
10855 /* Base and size of a partition in bytes. */
10856 struct device {
10857     u64_t dv_base;
10858     u64_t dv_size;
10859 };
10860
```

```

10861 #define NIL_DEV ((struct device *) 0)
10862
10863 /* Functions defined by driver.c */
10864 _PROTOTYPE( void driver_task, (struct driver *dr) );
10865 _PROTOTYPE( char *no_name, (void) );
10866 _PROTOTYPE( int do_nop, (struct driver *dp, message *m_ptr) );
10867 _PROTOTYPE( struct device *nop_prepare, (int device) );
10868 _PROTOTYPE( void nop_cleanup, (void) );
10869 _PROTOTYPE( void nop_task, (void) );
10870 _PROTOTYPE( void nop_signal, (struct driver *dp, message *m_ptr) );
10871 _PROTOTYPE( void nop_alarm, (struct driver *dp, message *m_ptr) );
10872 _PROTOTYPE( int nop_cancel, (struct driver *dp, message *m_ptr) );
10873 _PROTOTYPE( int nop_select, (struct driver *dp, message *m_ptr) );
10874 _PROTOTYPE( int do_diocnt1, (struct driver *dp, message *m_ptr) );
10875
10876 /* Parameters for the disk drive. */
10877 #define SECTOR_SIZE 512 /* physical sector size in bytes */
10878 #define SECTOR_SHIFT 9 /* for division */
10879 #define SECTOR_MASK 511 /* and remainder */
10880
10881 /* Size of the DMA buffer buffer in bytes. */
10882 #define USE_EXTRA_DMA_BUF 0 /* usually not needed */
10883 #define DMA_BUF_SIZE (DMA_SECTORS * SECTOR_SIZE)
10884
10885 #if (CHIP == INTEL)
10886 extern u8_t *tmp_buf; /* the DMA buffer */
10887 #else
10888 extern u8_t tmp_buf[]; /* the DMA buffer */
10889 #endif
10890 extern phys_bytes tmp_phys; /* phys address of DMA buffer */

```

+++++  
drivers/libdriver/drvl.h  
+++++

```

10900 /* IBM device driver definitions
10901 *
10902 */
10903
10904 #include <ibm/partition.h>
10905
10906 _PROTOTYPE( void partition, (struct driver *dr, int device, int style, int atapi) );
10907
10908 /* BIOS parameter table layout. */
10909 #define bp_cylinders(t) (* (u16_t *) (&(t)[0]))
10910 #define bp_heads(t) (* (u8_t *) (&(t)[2]))
10911 #define bp_reduced_wr(t) (* (u16_t *) (&(t)[3]))
10912 #define bp_precomp(t) (* (u16_t *) (&(t)[5]))
10913 #define bp_max_ecc(t) (* (u8_t *) (&(t)[7]))
10914 #define bp_ctlbyte(t) (* (u8_t *) (&(t)[8]))
10915 #define bp_landingzone(t) (* (u16_t *) (&(t)[12]))
10916 #define bp_sectors(t) (* (u8_t *) (&(t)[14]))
10917
10918 /* Miscellaneous. */
10919 #define DEV_PER_DRIVE (1 + NR_PARTITIONS)
10920 #define MINOR_t0 64
10921 #define MINOR_r0 120
10922 #define MINOR_d0p0s0 128
10923 #define MINOR_fd0p0 (28<<2)
10924 #define P_FLOPPY 0
10925 #define P_PRIMARY 1
10926 #define P_SUB 2

```

```
+++++
drivers/libdriver/driver.c
+++++

11000 /* This file contains device independent device driver interface.
11001 *
11002 * Changes:
11003 * Jul 25, 2005 added SYS_SIG type for signals (Jorrit N. Herder)
11004 * Sep 15, 2004 added SYN_ALARM type for timeouts (Jorrit N. Herder)
11005 * Jul 23, 2004 removed kernel dependencies (Jorrit N. Herder)
11006 * Apr 02, 1992 constructed from AT wini and floppy driver (Kees J. Bot)
11007 *
11008 *
11009 * The drivers support the following operations (using message format m2):
11010 *
11011 * m_type DEVICE PROC_NR COUNT POSITION ADDRESS
11012 *
11013 * | DEV_OPEN | device | proc nr | | | |
11014 * |-----+-----+-----+-----+-----+-----|
11015 * | DEV_CLOSE | device | proc nr | | | |
11016 * |-----+-----+-----+-----+-----+-----|
11017 * | DEV_READ | device | proc nr | bytes | offset | buf ptr
11018 * |-----+-----+-----+-----+-----+-----|
11019 * | DEV_WRITE | device | proc nr | bytes | offset | buf ptr
11020 * |-----+-----+-----+-----+-----+-----|
11021 * | DEV_GATHER | device | proc nr | iov len | offset | iov ptr
11022 * |-----+-----+-----+-----+-----+-----|
11023 * | DEV_SCATTER | device | proc nr | iov len | offset | iov ptr
11024 * |-----+-----+-----+-----+-----+-----|
11025 * | DEV_IOCTL | device | proc nr | func code | buf ptr
11026 * |-----+-----+-----+-----+-----+-----|
11027 * | CANCEL | device | proc nr | r/w | | |
11028 * |-----+-----+-----+-----+-----+-----|
11029 * | HARD_STOP | | | | | |
11030 *
11031 *
11032 * The file contains one entry point:
11033 *
11034 * driver_task: called by the device dependent task entry
11035 */
11036
11037 #include "../drivers.h"
11038 #include <sys/ioc_disk.h>
11039 #include "driver.h"
11040
11041 #define BUF_EXTRA 0
11042
11043 /* Claim space for variables. */
11044 PRIVATE u8_t buffer[(unsigned) 2 * DMA_BUF_SIZE + BUF_EXTRA];
11045 u8_t *tmp_buf; /* the DMA buffer eventually */
11046 phys_bytes tmp_phys; /* phys address of DMA buffer */
11047
11048 FORWARD _PROTOTYPE( void init_buffer, (void) );
11049 FORWARD _PROTOTYPE( int do_rdwrt, (struct driver *dr, message *mp) );
11050 FORWARD _PROTOTYPE( int do_vrdwrt, (struct driver *dr, message *mp) );
11051
11052 int device_caller;
11053
11054 /*=====
11055 * driver_task
11056 *=====
11057 PUBLIC void driver_task(dp)
11058 struct driver *dp; /* Device dependent entry points. */
11059 {
11060 /* Main program of any device driver task. */
11061
11062 int r, proc_nr;
```

```

11063     message mess;
11064
11065     /* Get a DMA buffer. */
11066     init_buffer();
11067
11068     /* Here is the main loop of the disk task. It waits for a message, carries
11069     * it out, and sends a reply.
11070     */
11071     while (TRUE) {
11072
11073         /* Wait for a request to read or write a disk block. */
11074         if(receive(ANY, &mess) != OK) continue;
11075
11076         device_caller = mess.m_source;
11077         proc_nr = mess.PROC_NR;
11078
11079         /* Now carry out the work. */
11080         switch(mess.m_type) {
11081             case DEV_OPEN:      r = (*dp->dr_open)(dp, &mess); break;
11082             case DEV_CLOSE:    r = (*dp->dr_close)(dp, &mess); break;
11083             case DEV_IOCTL:   r = (*dp->dr_ioctl)(dp, &mess); break;
11084             case CANCEL:       r = (*dp->dr_cancel)(dp, &mess);break;
11085             case DEV_SELECT:  r = (*dp->dr_select)(dp, &mess);break;
11086
11087             case DEV_READ:     r = do_rdwt(dp, &mess);      break;
11088             case DEV_WRITE:    r = do_rdwt(dp, &mess);      break;
11089             case DEV_GATHER:   r = do_vrdwt(dp, &mess);     break;
11090             case DEV_SCATTER:  r = do_vrdwt(dp, &mess);     break;
11091
11092             case HARD_INT:     /* leftover interrupt or expired timer. */
11093                 if(dp->dr_hw_int) {
11094                     (*dp->dr_hw_int)(dp, &mess);
11095                 }
11096                 continue;
11097             case SYS_SIG:       (*dp->dr_signal)(dp, &mess);
11098                 continue; /* don't reply */
11099             case SYN_ALARM:    (*dp->dr_alarm)(dp, &mess);
11100                 continue; /* don't reply */
11101             default:
11102                 if(dp->dr_other)
11103                     r = (*dp->dr_other)(dp, &mess);
11104                 else
11105                     r = EINVAL;
11106                 break;
11107             }
11108
11109         /* Clean up leftover state. */
11110         (*dp->dr_cleanup)();
11111
11112         /* Finally, prepare and send the reply message. */
11113         if (r != EDONTREPLY) {
11114             mess.m_type = TASK_REPLY;
11115             mess.REP_PROC_NR = proc_nr;
11116             /* Status is # of bytes transferred or error code. */
11117             mess.REP_STATUS = r;
11118             send(device_caller, &mess);
11119         }
11120     }
11121 }

11123 /*=====
11124 *          init_buffer
11125 *=====
11126 PRIVATE void init_buffer()
11127 {
11128     /* Select a buffer that can safely be used for DMA transfers. It may also
11129     * be used to read partition tables and such. Its absolute address is

```

```

11130 * 'tmp_phys', the normal address is 'tmp_buf'.
11131 */
11132 unsigned left;
11134
11135 tmp_buf = buffer;
11136 sys_umap(SELF, D, (vir_bytes)buffer, (phys_bytes)sizeof(buffer), &tmp_phys);
11137
11138 if ((left = dma_bytes_left(tmp_phys)) < DMA_BUF_SIZE) {
11139     /* First half of buffer crosses a 64K boundary, can't DMA into that */
11140     tmp_buf += left;
11141     tmp_phys += left;
11142 }
11143 }

11145 /*=====
11146 *          do_rdw
11147 *=====
11148 PRIVATE int do_rdw(dp, mp)
11149 struct driver *dp;           /* device dependent entry points */
11150 message *mp;                /* pointer to read or write message */
11151 {
11152 /* Carry out a single read or write request. */
11153 iovec_t iovecl;
11154 int r, opcode;
11155 phys_bytes phys_addr;
11156
11157 /* Disk address? Address and length of the user buffer? */
11158 if (mp->COUNT < 0) return(EINVAL);
11159
11160 /* Check the user buffer. */
11161 sys_umap(mp->PROC_NR, D, (vir_bytes) mp->ADDRESS, mp->COUNT, &phys_addr);
11162 if (phys_addr == 0) return(EFAULT);
11163
11164 /* Prepare for I/O. */
11165 if ((*dp->dr_prepare)(mp->DEVICE) == NIL_DEV) return(ENXIO);
11166
11167 /* Create a one element scatter/gather vector for the buffer. */
11168 opcode = mp->m_type == DEV_READ ? DEV_GATHER : DEV_SCATTER;
11169 iovecl.iov_addr = (vir_bytes) mp->ADDRESS;
11170 iovecl.iov_size = mp->COUNT;
11171
11172 /* Transfer bytes from/to the device. */
11173 r = (*dp->dr_transfer)(mp->PROC_NR, opcode, mp->POSITION, &iovecl, 1);
11174
11175 /* Return the number of bytes transferred or an error code. */
11176 return(r == OK ? (mp->COUNT - iovecl.iov_size) : r);
11177 }

11179 /*=====
11180 *          do_vrdw
11181 *=====
11182 PRIVATE int do_vrdw(dp, mp)
11183 struct driver *dp;           /* device dependent entry points */
11184 message *mp;                /* pointer to read or write message */
11185 {
11186 /* Carry out an device read or write to/from a vector of user addresses.
11187 * The "user addresses" are assumed to be safe, i.e. FS transferring to/from
11188 * its own buffers, so they are not checked.
11189 */
11190 static iovec_t iovec[NR_IOREQS];
11191 iovec_t *iov;
11192 phys_bytes iovec_size;
11193 unsigned nr_req;
11194 int r;
11195

```

```

11196     nr_req = mp->COUNT; /* Length of I/O vector */
11197
11198     if (mp->m_source < 0) {
11199         /* Called by a task, no need to copy vector. */
11200         iov = (iovec_t *) mp->ADDRESS;
11201     } else {
11202         /* Copy the vector from the caller to kernel space. */
11203         if (nr_req > NR_IOREQS) nr_req = NR_IOREQS;
11204         iovec_size = (phys_bytes) (nr_req * sizeof(iovec[0]));
11205
11206         if (OK != sys_datacopy(mp->m_source, (vir_bytes) mp->ADDRESS,
11207                                 SELF, (vir_bytes) iovec, iovec_size))
11208             panic((*dp->dr_name)(), "bad I/O vector by", mp->m_source);
11209         iov = iovec;
11210     }
11211
11212     /* Prepare for I/O. */
11213     if ((*dp->dr_prepare)(mp->DEVICE) == NIL_DEV) return(ENXIO);
11214
11215     /* Transfer bytes from/to the device. */
11216     r = (*dp->dr_transfer)(mp->PROC_NR, mp->m_type, mp->POSITION, iov, nr_req);
11217
11218     /* Copy the I/O vector back to the caller. */
11219     if (mp->m_source >= 0) {
11220         sys_datacopy(SELF, (vir_bytes) iovec,
11221                     mp->m_source, (vir_bytes) mp->ADDRESS, iovec_size);
11222     }
11223     return(r);
11224 }

11226 /*=====
11227 *          no_name
11228 *=====
11229 PUBLIC char *no_name()
11230 {
11231 /* Use this default name if there is no specific name for the device. This was
11232 * originally done by fetching the name from the task table for this process:
11233 * "return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);", but currently a
11234 * real "noname" is returned. Perhaps, some system information service can be
11235 * queried for a name at a later time.
11236 */
11237     static char name[] = "noname";
11238     return name;
11239 }

11241 /*=====
11242 *          do_nop
11243 *=====
11244 PUBLIC int do_nop(dp, mp)
11245 struct driver *dp;
11246 message *mp;
11247 {
11248 /* Nothing there, or nothing to do. */
11249
11250     switch (mp->m_type) {
11251     case DEV_OPEN:      return(ENODEV);
11252     case DEV_CLOSE:    return(OK);
11253     case DEV_IOCTL:    return(ENOTTY);
11254     default:           return(EIO);
11255     }
11256 }

11258 /*=====
11259 *          nop_signal
11260 *=====
11261 PUBLIC void nop_signal(dp, mp)
11262 struct driver *dp;

```

```

11263 message *mp;
11264 {
11265 /* Default action for signal is to ignore. */
11266 }
11267 /*=====
11268 *          nop_alarm
11269 *=====
11270 PUBLIC void nop_alarm(dp, mp)
11271 struct driver *dp;
11272 message *mp;
11273 {
11274 /* Ignore the leftover alarm. */
11275 }
11276 */
11277 /*=====
11278 *          nop_prepare
11279 *=====
11280 PUBLIC struct device *nop_prepare(device)
11281 {
11282 /* Nothing to prepare for. */
11283 return(NIL_DEV);
11284 }
11285 */
11286 /*=====
11287 *          nop_cleanup
11288 *=====
11289 PUBLIC void nop_cleanup()
11290 {
11291 /* Nothing to clean up. */
11292 }
11293 */
11294 /*=====
11295 *          nop_cancel
11296 *=====
11297 PUBLIC int nop_cancel(struct driver *dr, message *m)
11298 {
11299 /* Nothing to do for cancel. */
11300 return(OK);
11301 }
11302 */
11303 /*=====
11304 *          nop_select
11305 *=====
11306 PUBLIC int nop_select(struct driver *dr, message *m)
11307 {
11308 /* Nothing to do for select. */
11309 return(OK);
11310 }
11311 */
11312 /*=====
11313 *          do_ioctl
11314 *=====
11315 PUBLIC int do_ioctl(dp, mp)
11316 struct driver *dp;
11317 message *mp;           /* pointer to ioctl request */
11318 {
11319 /* Carry out a partition setting/getting request. */
11320     struct device *dv;
11321     struct partition entry;
11322     int s;
11323
11324     if (mp->REQUEST != DIOCSETP && mp->REQUEST != DIOCGGETP) {
11325         if(dp->dr_other) {
11326             return dp->dr_other(dp, mp);
11327         } else return(ENOTTY);
11328     }
11329 }
```

```

11330
11331 /* Decode the message parameters. */
11332 if ((dv = (*dp->dr_prepare)(mp->DEVICE)) == NIL_DEV) return(ENXIO);
11333
11334 if (mp->REQUEST == DIOCSETP) {
11335     /* Copy just this one partition table entry. */
11336     if (OK != (s=sys_datacopy(mp->PROC_NR, (vir_bytes) mp->ADDRESS,
11337                         SELF, (vir_bytes) &entry, sizeof(entry))))
11338         return s;
11339     dv->dv_base = entry.base;
11340     dv->dv_size = entry.size;
11341 } else {
11342     /* Return a partition table entry and the geometry of the drive. */
11343     entry.base = dv->dv_base;
11344     entry.size = dv->dv_size;
11345     (*dp->dr_geometry)(&entry);
11346     if (OK != (s=sys_datacopy(SELF, (vir_bytes) &entry,
11347                         mp->PROC_NR, (vir_bytes) mp->ADDRESS, sizeof(entry))))
11348         return s;
11349 }
11350 return(OK);
11351 }

+++++
drivers/libdriver/drvlib.c
+++++
11400 /* IBM device driver utility functions.          Author: Kees J. Bot
11401 *                                                 7 Dec 1995
11402 * Entry point:
11403 *   partition: partition a disk to the partition table(s) on it.
11404 */
11405
11406 #include "driver.h"
11407 #include "drvlib.h"
11408 #include <unistd.h>
11409
11410 /* Extended partition? */
11411 #define ext_part(s) ((s) == 0x05 || (s) == 0x0F)
11412
11413 FORWARD _PROTOTYPE( void extpartition, (struct driver *dp, int extdev,
11414                           unsigned long extbase) );
11415 FORWARD _PROTOTYPE( int get_part_table, (struct driver *dp, int device,
11416                           unsigned long offset, struct part_entry *table));
11417 FORWARD _PROTOTYPE( void sort, (struct part_entry *table) );
11418
11419 #ifndef CD_SECTOR_SIZE
11420 #define CD_SECTOR_SIZE 2048
11421 #endif
11422
11423 /*=====
11424 *           partition
11425 *=====
11426 PUBLIC void partition(dp, device, style, atapi)
11427 struct driver *dp;      /* device dependent entry points */
11428 int device;            /* device to partition */
11429 int style;              /* partitioning style: floppy, primary, sub. */
11430 int atapi;             /* atapi device */
11431 {
11432 /* This routine is called on first open to initialize the partition tables
11433 * of a device. It makes sure that each partition falls safely within the
11434 * device's limits. Depending on the partition style we are either making
11435 * floppy partitions, primary partitions or subpartitions. Only primary
11436 * partitions are sorted, because they are shared with other operating
11437 * systems that expect this.
11438 */

```

```

11439 struct part_entry table[NR_PARTITIONS], *pe;
11440 int disk, par;
11441 struct device *dv;
11442 unsigned long base, limit, part_limit;
11443
11444 /* Get the geometry of the device to partition */
11445 if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV
11446           || cmp64u(dv->dv_size, 0) == 0) return;
11447 base = div64u(dv->dv_base, SECTOR_SIZE);
11448 limit = base + div64u(dv->dv_size, SECTOR_SIZE);
11449
11450 /* Read the partition table for the device. */
11451 if(!get_part_table(dp, device, 0L, table)) {
11452     return;
11453 }
11454
11455 /* Compute the device number of the first partition. */
11456 switch (style) {
11457 case P_FLOPPY:
11458     device += MINOR_fd0p0;
11459     break;
11460 case P_PRIMARY:
11461     sort(table);           /* sort a primary partition table */
11462     device += 1;
11463     break;
11464 case P_SUB:
11465     disk = device / DEV_PER_DRIVE;
11466     par = device % DEV_PER_DRIVE - 1;
11467     device = MINOR_d0p0s0 + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
11468 }
11469
11470 /* Find an array of devices. */
11471 if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV) return;
11472
11473 /* Set the geometry of the partitions from the partition table. */
11474 for (par = 0; par < NR_PARTITIONS; par++, dv++) {
11475     /* Shrink the partition to fit within the device. */
11476     pe = &table[par];
11477     part_limit = pe->lowsec + pe->size;
11478     if (part_limit < pe->lowsec) part_limit = limit;
11479     if (part_limit > limit) part_limit = limit;
11480     if (pe->lowsec < base) pe->lowsec = base;
11481     if (part_limit < pe->lowsec) part_limit = pe->lowsec;
11482
11483     dv->dv_base = mul64u(pe->lowsec, SECTOR_SIZE);
11484     dv->dv_size = mul64u(part_limit - pe->lowsec, SECTOR_SIZE);
11485
11486     if (style == P_PRIMARY) {
11487         /* Each Minix primary partition can be subpartitioned. */
11488         if (pe->sysind == MINIX_PART)
11489             partition(dp, device + par, P_SUB, atapi);
11490
11491         /* An extended partition has logical partitions. */
11492         if (ext_part(pe->sysind))
11493             extpartition(dp, device + par, pe->lowsec);
11494     }
11495 }
11496 }

11497 /*=====
11498      *                               extpartition
11499      *=====
11500 PRIVATE void extpartition(dp, extdev, extbase)
11501 struct driver *dp;      /* device dependent entry points */
11502 int extdev;            /* extended partition to scan */
11503 unsigned long extbase; /* sector offset of the base extended partition */
11504 {

```

```

11506 /* Extended partitions cannot be ignored alas, because people like to move
11507 * files to and from DOS partitions. Avoid reading this code, it's no fun.
11508 */
11509 struct part_entry table[NR_PARTITIONS], *pe;
11510 int subdev, disk, par;
11511 struct device *dv;
11512 unsigned long offset, nextoffset;
11513
11514 disk = extdev / DEV_PER_DRIVE;
11515 par = extdev % DEV_PER_DRIVE - 1;
11516 subdev = MINOR_d0p0s0 + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
11517
11518 offset = 0;
11519 do {
11520     if (!get_part_table(dp, extdev, offset, table)) return;
11521     sort(table);
11522
11523     /* The table should contain one logical partition and optionally
11524      * another extended partition. (It's a linked list.)
11525      */
11526     nextoffset = 0;
11527     for (par = 0; par < NR_PARTITIONS; par++) {
11528         pe = &table[par];
11529         if (ext_part(pe->sysind)) {
11530             nextoffset = pe->lowsec;
11531         } else
11532             if (pe->sysind != NO_PART) {
11533                 if ((dv = (*dp->dr_prepare)(subdev)) == NIL_DEV) return;
11534
11535                 dv->dv_base = mul64u(extbase + offset + pe->lowsec,
11536                                         SECTOR_SIZE);
11537                 dv->dv_size = mul64u(pe->size, SECTOR_SIZE);
11538
11539                 /* Out of devices? */
11540                 if (++subdev % NR_PARTITIONS == 0) return;
11541             }
11542         }
11543     } while ((offset = nextoffset) != 0);
11544 }

11546 /*=====
11547 *          get_part_table
11548 *=====
11549 PRIVATE int get_part_table(dp, device, offset, table)
11550 struct driver *dp;
11551 int device;
11552 unsigned long offset;           /* sector offset to the table */
11553 struct part_entry *table;      /* four entries */
11554 {
11555 /* Read the partition table for the device, return true iff there were no
11556 * errors.
11557 */
11558 iovec_t iovecl;
11559 off_t position;
11560 static unsigned char partbuf[CD_SECTOR_SIZE];
11561
11562 position = offset << SECTOR_SHIFT;
11563 iovecl.iov_addr = (vir_bytes) partbuf;
11564 iovecl.iov_size = CD_SECTOR_SIZE;
11565 if ((*dp->dr_prepare)(device) != NIL_DEV) {
11566     (void) (*dp->dr_transfer)(SELF, DEV_GATHER, position, &iovec1, 1);
11567 }
11568 if (iovec1.iov_size != 0) {
11569     return 0;
11570 }
11571 if (partbuf[510] != 0x55 || partbuf[511] != 0xAA) {
11572     /* Invalid partition table. */
11573     return 0;

```

```

11574     }
11575     memcpy(table, (partbuf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
11576     return 1;
11577 }

11579 /*=====
11580 *          sort
11581 *=====
11582 PRIVATE void sort(table)
11583 struct part_entry *table;
11584 {
11585 /* Sort a partition table. */
11586     struct part_entry *pe, tmp;
11587     int n = NR_PARTITIONS;
11588
11589     do {
11590         for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
11591             if (pe[0].sysind == NO_PART
11592                 || (pe[0].lowsec > pe[1].lowsec
11593                     && pe[1].sysind != NO_PART)) {
11594                 tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
11595             }
11596         }
11597     } while (--n > 0);
11598 }

```

```
+++++
drivers/memory/memory.c
+++++
```

```

11600 /* This file contains the device dependent part of the drivers for the
11601 * following special files:
11602 *   /dev/ram      - RAM disk
11603 *   /dev/mem      - absolute memory
11604 *   /dev/kmem     - kernel virtual memory
11605 *   /dev/null     - null device (data sink)
11606 *   /dev/boot     - boot device loaded from boot image
11607 *   /dev/zero     - null byte stream generator
11608 *
11609 * Changes:
11610 *   Apr 29, 2005  added null byte generator (Jorrit N. Herder)
11611 *   Apr 09, 2005  added support for boot device (Jorrit N. Herder)
11612 *   Jul 26, 2004  moved RAM driver to user-space (Jorrit N. Herder)
11613 *   Apr 20, 1992  device dependent/independent split (Kees J. Bot)
11614 */
11615
11616 #include "../drivers.h"
11617 #include "../libdriver/driver.h"
11618 #include <sys/ioc_memory.h>
11619 #include "../kernel/const.h"
11620 #include "../kernel/config.h"
11621 #include "../kernel/type.h"
11622
11623 #include "assert.h"
11624
11625 #define NR_DEVS          6           /* number of minor devices */
11626
11627 PRIVATE struct device m_geom[NR_DEVS]; /* base and size of each device */
11628 PRIVATE int m_seg[NR_DEVS];           /* segment index of each device */
11629 PRIVATE int m_device;                /* current device */
11630 PRIVATE struct kinfo kinfo;         /* kernel information */
11631 PRIVATE struct machine machine;    /* machine information */
11632
11633 extern int errno;                  /* error number for PM calls */
11634
11635 FORWARD _PROTOTYPE( char *m_name, (void) );

```

```

11636 FORWARD _PROTOTYPE( struct device *m_prepare, (int device) );
11637 FORWARD _PROTOTYPE( int m_transfer, (int proc_nr, int opcode, off_t position,
11638                                iovec_t *iov, unsigned nr_req) );
11639 FORWARD _PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr) );
11640 FORWARD _PROTOTYPE( void m_init, (void) );
11641 FORWARD _PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr) );
11642 FORWARD _PROTOTYPE( void m_geometry, (struct partition *entry) );
11643
11644 /* Entry points to this driver. */
11645 PRIVATE struct driver m_dtab = {
11646     m_name,          /* current device's name */
11647     m_do_open,        /* open or mount */
11648     do_nop,          /* nothing on a close */
11649     m_ioctl,          /* specify ram disk geometry */
11650     m_prepare,        /* prepare for I/O on a given minor device */
11651     m_transfer,       /* do the I/O */
11652     nop_cleanup,     /* no need to clean up */
11653     m_geometry,       /* memory device "geometry" */
11654     nop_signal,       /* system signals */
11655     nop_alarm,
11656     nop_cancel,
11657     nop_select,
11658     NULL,
11659     NULL
11660 };
11661
11662 /* Buffer for the /dev/zero null byte feed. */
11663 #define ZERO_BUF_SIZE           1024
11664 PRIVATE char dev_zero[ZERO_BUF_SIZE];
11665
11666 #define click_to_round_k(n) \
11667     ((unsigned) (((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))
11668
11669 /*=====
11670 *                         main
11671 *=====
11672 PUBLIC int main(void)
11673 {
11674     /* Main program. Initialize the memory driver and start the main loop. */
11675     m_init();
11676     driver_task(&m_dtab);
11677     return(OK);
11678 }
11679
11680 /*=====
11681 *                         m_name
11682 *=====
11683 PRIVATE char *m_name()
11684 {
11685     /* Return a name for the current device. */
11686     static char name[] = "memory";
11687     return name;
11688 }
11689
11690 /*=====
11691 *                         m_prepare
11692 *=====
11693 PRIVATE struct device *m_prepare(device)
11694 int device;
11695 {
11696     /* Prepare for I/O on a device: check if the minor device number is ok. */
11697     if (device < 0 || device >= NR_DEVS) return(NIL_DEV);
11698     m_device = device;
11699
11700     return(&m_geom[device]);
11701 }
```

```

11703 /*=====
11704 *          m_transfer
11705 *=====
11706 PRIVATE int m_transfer(proc_nr, opcode, position, iov, nr_req)
11707 int proc_nr;           /* process doing the request */
11708 int opcode;            /* DEV_GATHER or DEV_SCATTER */
11709 off_t position;       /* offset on device to read or write */
11710 iovec_t *iov;         /* pointer to read or write request vector */
11711 unsigned nr_req;      /* length of request vector */
11712 {
11713 /* Read or write one the driver's minor devices. */
11714 phys_bytes mem_phys;
11715 int seg;
11716 unsigned count, left, chunk;
11717 vir_bytes user_vir;
11718 struct device *dv;
11719 unsigned long dv_size;
11720 int s;
11721
11722 /* Get minor device number and check for /dev/null. */
11723 dv = &m_geom[m_device];
11724 dv_size = cv64ul(dv->dv_size);
11725
11726 while (nr_req > 0) {
11727
11728     /* How much to transfer and where to / from. */
11729     count = iov->iov_size;
11730     user_vir = iov->iov_addr;
11731
11732     switch (m_device) {
11733
11734         /* No copying; ignore request. */
11735         case NULL_DEV:
11736             if (opcode == DEV_GATHER) return(OK);      /* always at EOF */
11737             break;
11738
11739         /* Virtual copying. For RAM disk, kernel memory and boot device. */
11740         case RAM_DEV:
11741         case KMEM_DEV:
11742         case BOOT_DEV:
11743             if (position >= dv_size) return(OK);      /* check for EOF */
11744             if (position + count > dv_size) count = dv_size - position;
11745             seg = m_seg[m_device];
11746
11747             if (opcode == DEV_GATHER) {                /* copy actual data */
11748                 sys_vircopy(SELF,seg,position, proc_nr,D,user_vir, count);
11749             } else {
11750                 sys_vircopy(proc_nr,D,user_vir, SELF,seg,position, count);
11751             }
11752             break;
11753
11754         /* Physical copying. Only used to access entire memory. */
11755         case MEM_DEV:
11756             if (position >= dv_size) return(OK);      /* check for EOF */
11757             if (position + count > dv_size) count = dv_size - position;
11758             mem_phys = cv64ul(dv->dv_base) + position;
11759
11760             if (opcode == DEV_GATHER) {                /* copy data */
11761                 sys_physcopy(NONE, PHYS_SEG, mem_phys,
11762                             proc_nr, D, user_vir, count);
11763             } else {
11764                 sys_physcopy(proc_nr, D, user_vir,
11765                             NONE, PHYS_SEG, mem_phys, count);
11766             }
11767             break;
11768
11769         /* Null byte stream generator. */

```

```

11770         case ZERO_DEV:
11771             if (opcode == DEV_GATHER) {
11772                 left = count;
11773                 while (left > 0) {
11774                     chunk = (left > ZERO_BUF_SIZE) ? ZERO_BUF_SIZE : left;
11775                     if (OK != (s=sys_vircopy(SELF, D, (vir_bytes) dev_zero,
11776                                     proc_nr, D, user_vir, chunk)))
11777                         report("MEM","sys_vircopy failed", s);
11778                     left -= chunk;
11779                     user_vir += chunk;
11780                 }
11781             }
11782             break;
11783
11784         /* Unknown (illegal) minor device. */
11785     default:
11786         return(EINVAL);
11787     }
11788
11789     /* Book the number of bytes transferred. */
11790     position += count;
11791     iov->iov_addr += count;
11792     if ((iov->iov_size -= count) == 0) { iov++; nr_req--; }
11793
11794 }
11795     return(OK);
11796 }

11798 /*=====
11799 *           m_do_open
11800 *=====
11801 PRIVATE int m_do_open(dp, m_ptr)
11802 struct driver *dp;
11803 message *m_ptr;
11804 {
11805 /* Check device number on open. (This used to give I/O privileges to a
11806 * process opening /dev/mem or /dev/kmem. This may be needed in case of
11807 * memory mapped I/O. With system calls to do I/O this is no longer needed.)
11808 */
11809     if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
11810
11811     return(OK);
11812 }

11814 /*=====
11815 *           m_init
11816 *=====
11817 PRIVATE void m_init()
11818 {
11819     /* Initialize this task. All minor devices are initialized one by one. */
11820     int i, s;
11821
11822     if (OK != (s=sys_getkinfo(&kinfo))) {
11823         panic("MEM","Couldn't get kernel information.",s);
11824     }
11825
11826     /* Install remote segment for /dev/kmem memory. */
11827     m_geom[KMEM_DEV].dv_base = cvl64(kinfo.kmem_base);
11828     m_geom[KMEM_DEV].dv_size = cvl64(kinfo.kmem_size);
11829     if (OK != (s=sys_segctl(&m_seg[KMEM_DEV], (u16_t *) &s, (vir_bytes *) &s,
11830                           kinfo.kmem_base, kinfo.kmem_size))) {
11831         panic("MEM","Couldn't install remote segment.",s);
11832     }
11833
11834     /* Install remote segment for /dev/boot memory, if enabled. */

```

```

11835     m_geom[BOOT_DEV].dv_base = cvu164(kinfo.bootdev_base);
11836     m_geom[BOOT_DEV].dv_size = cvu164(kinfo.bootdev_size);
11837     if (kinfo.bootdev_base > 0) {
11838         if (OK != (s=sys_segctl(&m_seg[BOOT_DEV], (u16_t *) &s, (vir_bytes *) &s,
11839                         kinfo.bootdev_base, kinfo.bootdev_size))) {
11840             panic("MEM","Couldn't install remote segment.",s);
11841         }
11842     }
11843
11844     /* Initialize /dev/zero. Simply write zeros into the buffer. */
11845     for (i=0; i<ZERO_BUF_SIZE; i++) {
11846         dev_zero[i] = '\0';
11847     }
11848
11849     /* Set up memory ranges for /dev/mem. */
11850     if (OK != (s=sys_getmachine(&machine))) {
11851         panic("MEM","Couldn't get machine information.",s);
11852     }
11853     if (! machine.protected) {
11854         m_geom[MEM_DEV].dv_size = cvu164(0x100000); /* 1M for 8086 systems */
11855     } else {
11856         m_geom[MEM_DEV].dv_size = cvu164(0xFFFFFFFF); /* 4G-1 for 386 systems */
11857     }
11858 }

11860 /*=====
11861 *          m_ioctl
11862 *=====
11863 PRIVATE int m_ioctl(dp, m_ptr)
11864 struct driver *dp;                                /* pointer to driver structure */
11865 message *m_ptr;                                  /* pointer to control message */
11866 {
11867     /* I/O controls for the memory driver. Currently there is one I/O control:
11868     * - MIOCRAMSIZE: to set the size of the RAM disk.
11869     */
11870     struct device *dv;
11871     if ((dv = m_prepare(m_ptr->DEVICE)) == NIL_DEV) return(ENXIO);
11872
11873     switch (m_ptr->REQUEST) {
11874         case MIOCRAMSIZE: {
11875             /* FS wants to create a new RAM disk with the given size. */
11876             phys_bytes ramdev_size;
11877             phys_bytes ramdev_base;
11878             int s;
11879
11880             if (m_ptr->PROC_NR != FS_PROC_NR) {
11881                 report("MEM", "warning, MIOCRAMSIZE called by", m_ptr->PROC_NR);
11882                 return(EPERM);
11883             }
11884
11885             /* Try to allocate a piece of memory for the RAM disk. */
11886             ramdev_size = m_ptr->POSITION;
11887             if (allocmem(ramdev_size, &ramdev_base) < 0) {
11888                 report("MEM", "warning, allocmem failed", errno);
11889                 return(ENOMEM);
11890             }
11891             dv->dv_base = cvu164(ramdev_base);
11892             dv->dv_size = cvu164(ramdev_size);
11893
11894             if (OK != (s=sys_segctl(&m_seg[RAM_DEV], (u16_t *) &s, (vir_bytes *) &s,
11895                             ramdev_base, ramdev_size))) {
11896                 panic("MEM","Couldn't install remote segment.",s);
11897             }
11898             break;
11899         }
11900
11901     default:
11902         return(do_diocntl(&m_dtab, m_ptr));

```

```

11903     }
11904     return(OK);
11905 }

11907 /*=====
11908 *          m_geometry
11909 *=====
11910 PRIVATE void m_geometry(entry)
11911 struct partition *entry;
11912 {
11913     /* Memory devices don't have a geometry, but the outside world insists. */
11914     entry->cylinders = div64u(m_geom[m_device].dv_size, SECTOR_SIZE) / (64 * 32);
11915     entry->heads = 64;
11916     entry->sectors = 32;
11917 }

+++++
drivers/at_wini/at_wini.h
+++++

12000 #include "../drivers.h"
12001 #include "../libdriver/driver.h"
12002 #include "../libdriver/drplib.h"
12003
12004 _PROTOTYPE(int main, (void));
12005
12006 #define VERBOSE          0      /* display identify messages during boot */
12007 #define ENABLE_ATAPI      0      /* add ATAPI cd-rom support to driver */

+++++
drivers/at_wini/at_wini.c
+++++

12100 /* This file contains the device dependent part of a driver for the IBM-AT
12101 * winchester controller. Written by Adri Koppes.
12102 *
12103 * The file contains one entry point:
12104 *
12105 *   at_winchester_task:      main entry when system is brought up
12106 *
12107 * Changes:
12108 *   Aug 19, 2005  ata pci support, supports SATA (Ben Gras)
12109 *   Nov 18, 2004  moved AT disk driver to user-space (Jorrit N. Herder)
12110 *   Aug 20, 2004  watchdogs replaced by sync alarms (Jorrit N. Herder)
12111 *   Mar 23, 2000  added ATAPI CDROM support (Michael Temari)
12112 *   May 14, 2000  d-d-i rewrite (Kees J. Bot)
12113 *   Apr 13, 1992  device dependent/independent split (Kees J. Bot)
12114 */
12115
12116 #include "at_wini.h"
12117 #include "../libpci/pci.h"
12118
12119 #include <minix/sysutil.h>
12120 #include <minix/keymap.h>
12121 #include <sys/ioc_disk.h>
12122
12123 #define ATAPI_DEBUG          0      /* To debug ATAPI code. */
12124
12125 /* I/O Ports used by winchester disk controllers. */
12126
12127 /* Read and write registers */
12128 #define REG_CMD_BASE0  0x1F0  /* command base register of controller 0 */
12129 #define REG_CMD_BASE1  0x170  /* command base register of controller 1 */
12130 #define REG_CTL_BASE0  0x3F6  /* control base register of controller 0 */

```

```

12131 #define REG_CTL_BASE1 0x376 /* control base register of controller 1 */
12132
12133 #define REG_DATA 0 /* data register (offset from the base reg.) */
12134 #define REG_PRECOMP 1 /* start of write precompensation */
12135 #define REG_COUNT 2 /* sectors to transfer */
12136 #define REG_SECTOR 3 /* sector number */
12137 #define REG_CYL_LO 4 /* low byte of cylinder number */
12138 #define REG_CYL_HI 5 /* high byte of cylinder number */
12139 #define REG_LDH 6 /* lba, drive and head */
12140 #define LDH_DEFAULT 0xA0 /* ECC enable, 512 bytes per sector */
12141 #define LDH_LBA 0x40 /* Use LBA addressing */
12142 #define ldh_init(drive) (LDH_DEFAULT | ((drive) << 4))
12143
12144 /* Read only registers */
12145 #define REG_STATUS 7 /* status */
12146 #define STATUS_BSY 0x80 /* controller busy */
12147 #define STATUS_RDY 0x40 /* drive ready */
12148 #define STATUS_WF 0x20 /* write fault */
12149 #define STATUS_SC 0x10 /* seek complete (obsolete) */
12150 #define STATUS_DRQ 0x08 /* data transfer request */
12151 #define STATUS_CRD 0x04 /* corrected data */
12152 #define STATUS_IDX 0x02 /* index pulse */
12153 #define STATUS_ERR 0x01 /* error */
12154 #define STATUS_ADMBSY 0x100 /* administratively busy (software) */
12155 #define REG_ERROR 1 /* error code */
12156 #define ERROR_BB 0x80 /* bad block */
12157 #define ERROR_ECC 0x40 /* bad ecc bytes */
12158 #define ERROR_ID 0x10 /* id not found */
12159 #define ERROR_AC 0x04 /* aborted command */
12160 #define ERROR_TK 0x02 /* track zero error */
12161 #define ERROR_DM 0x01 /* no data address mark */
12162
12163 /* Write only registers */
12164 #define REG_COMMAND 7 /* command */
12165 #define CMD_IDLE 0x00 /* for w_command: drive idle */
12166 #define CMD_RECALIBRATE 0x10 /* recalibrate drive */
12167 #define CMD_READ 0x20 /* read data */
12168 #define CMD_READ_EXT 0x24 /* read data (LBA48 addressed) */
12169 #define CMD_WRITE 0x30 /* write data */
12170 #define CMD_WRITE_EXT 0x34 /* write data (LBA48 addressed) */
12171 #define CMD_READVERIFY 0x40 /* read verify */
12172 #define CMD_FORMAT 0x50 /* format track */
12173 #define CMD_SEEK 0x70 /* seek cylinder */
12174 #define CMD_DIAG 0x90 /* execute device diagnostics */
12175 #define CMD_SPECIFY 0x91 /* specify parameters */
12176 #define ATA_IDENTIFY 0xEC /* identify drive */
12177 /* #define REG_CTL 0x206 /* control register */
12178 #define REG_CTL 0 /* control register */
12179 #define CTL_NORETRY 0x80 /* disable access retry */
12180 #define CTL_NOECC 0x40 /* disable ecc retry */
12181 #define CTL_EIGHTHEADS 0x08 /* more than eight heads */
12182 #define CTL_RESET 0x04 /* reset controller */
12183 #define CTL_INTDISABLE 0x02 /* disable interrupts */
12184
12185 #define REG_STATUS 7 /* status */
12186 #define STATUS_BSY 0x80 /* controller busy */
12187 #define STATUS_DRDY 0x40 /* drive ready */
12188 #define STATUS_DMAFD 0x20 /* dma ready/drive fault */
12189 #define STATUS_SRVDSC 0x10 /* service or dsc */
12190 #define STATUS_DRQ 0x08 /* data transfer request */
12191 #define STATUS_CORR 0x04 /* correctable error occurred */
12192 #define STATUS_CHECK 0x01 /* check error */
12193
12194 /* Interrupt request lines. */
12195 #define NO_IRQ 0 /* no IRQ set yet */
12196
12197 #define ATAPI_PACKETSIZE 12

```

```

12198 #define SENSE_PACKETSIZE      18
12199
12200 /* Common command block */
12201 struct command {
12202     u8_t precomp;          /* REG_PRECOMP, etc. */
12203     u8_t count;
12204     u8_t sector;
12205     u8_t cyl_lo;
12206     u8_t cyl_hi;
12207     u8_t ldh;
12208     u8_t command;
12209 };
12210
12211 /* Error codes */
12212 #define ERR           (-1)   /* general error */
12213 #define ERR_BAD_SECTOR (-2)   /* block marked bad detected */
12214
12215 /* Some controllers don't interrupt, the clock will wake us up. */
12216 #define WAKEUP        (32*HZ) /* drive may be out for 31 seconds max */
12217
12218 /* Miscellaneous. */
12219 #define MAX_DRIVES      8
12220 #define COMPAT_DRIVES    4
12221 #define MAX_SECS         256   /* controller can transfer this many sectors */
12222 #define MAX_ERRORS       4      /* how often to try rd/wt before quitting */
12223 #define NR_MINORS        (MAX_DRIVES * DEV_PER_DRIVE)
12224 #define SUB_PER_DRIVE     (NR_PARTITIONS * NR_PARTITIONS)
12225 #define NR_SUBDEVS        (MAX_DRIVES * SUB_PER_DRIVE)
12226 #define DELAY_USECS      1000  /* controller timeout in microseconds */
12227 #define DELAY_TICKS       1      /* controller timeout in ticks */
12228 #define DEF_TIMEOUT_TICKS 300   /* controller timeout in ticks */
12229 #define RECOVERY_USECS   500000 /* controller recovery time in microseconds */
12230 #define RECOVERY_TICKS   30    /* controller recovery time in ticks */
12231 #define INITIALIZED      0x01  /* drive is initialized */
12232 #define DEAF             0x02  /* controller must be reset */
12233 #define SMART            0x04  /* drive supports ATA commands */
12234 #define ATAPI            0      /* don't bother with ATAPI; optimise out */
12235 #define IDENTIFIED      0x10  /* w_identify done successfully */
12236 #define IGNORING         0x20  /* w_identify failed once */
12237
12238 /* Timeouts and max retries. */
12239 int timeout_ticks = DEF_TIMEOUT_TICKS, max_errors = MAX_ERRORS;
12240 int wakeup_ticks = WAKEUP;
12241 long w_standard_timeouts = 0, w_pci_debug = 0, w_instance = 0,
12242     w_lba48 = 0, atapi_debug = 0;
12243
12244 int w_testing = 0, w_silent = 0;
12245
12246 int w_next_drive = 0;
12247
12248 /* Variables. */
12249
12250 /* wini is indexed by controller first, then drive (0-3).
12251 * controller 0 is always the 'compatability' ide controller, at
12252 * the fixed locations, whether present or not.
12253 */
12254 PRIVATE struct wini {
12255     unsigned state;          /* main drive struct, one entry per drive */
12256     unsigned w_status;        /* drive state: deaf, initialized, dead */
12257     unsigned base_cmd;        /* device status register */
12258     unsigned base_ctl;        /* command base register */
12259     unsigned irq;            /* control base register */
12260     unsigned irq_mask;        /* interrupt request line */
12261     unsigned irq_need_ack;   /* 1 << irq */
12262     int irq_hook_id;         /* irq needs to be acknowledged */
12263     int lba48;               /* id of irq hook at the kernel */
12264     unsigned lcyliners;       /* supports lba48 */
12265     unsigned logical_cylinders; /* logical number of cylinders (BIOS) */

```

```

12265     unsigned lheads;          /* logical number of heads */
12266     unsigned lsectors;        /* logical number of sectors per track */
12267     unsigned pcylinders;      /* physical number of cylinders (translated) */
12268     unsigned pheads;          /* physical number of heads */
12269     unsigned psectors;        /* physical number of sectors per track */
12270     unsigned ldhpref;         /* top four bytes of the LDH (head) register */
12271     unsigned precomp;         /* write precompensation cylinder / 4 */
12272     unsigned max_count;       /* max request for this drive */
12273     unsigned open_ct;          /* in-use count */
12274     struct device part[DEV_PER_DRIVE]; /* disks and partitions */
12275     struct device subpart[SUB_PER_DRIVE]; /* subpartitions */
12276 } wini[MAX_DRIVES], *w_wn;
12277
12278 PRIVATE int w_device = -1;
12279 PRIVATE int w_controller = -1;
12280 PRIVATE int w_major = -1;
12281 PRIVATE char w_id_string[40];
12282
12283 PRIVATE int win_tasknr;           /* my task number */
12284 PRIVATE int w_command;           /* current command in execution */
12285 PRIVATE u8_t w_byteval;          /* used for SYS_IRQCTL */
12286 PRIVATE int w_drive;             /* selected drive */
12287 PRIVATE int w_controller;        /* selected controller */
12288 PRIVATE struct device *w_dv;    /* device's base and size */
12289
12290 FORWARD _PROTOTYPE( void init_params, (void) );
12291 FORWARD _PROTOTYPE( void init_drive, (struct wini *, int, int, int, int, int, int) );
12292 FORWARD _PROTOTYPE( void init_params_pci, (int) );
12293 FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
12294 FORWARD _PROTOTYPE( struct device *w_prepare, (int dev) );
12295 FORWARD _PROTOTYPE( int w_identify, (void) );
12296 FORWARD _PROTOTYPE( char *w_name, (void) );
12297 FORWARD _PROTOTYPE( int w_specify, (void) );
12298 FORWARD _PROTOTYPE( int w_io_test, (void) );
12299 FORWARD _PROTOTYPE( int w_transfer, (int proc_nr, int opcode, off_t position,
12300                           iovec_t *iov, unsigned nr_req) );
12301 FORWARD _PROTOTYPE( int com_out, (struct command *cmd) );
12302 FORWARD _PROTOTYPE( void w_need_reset, (void) );
12303 FORWARD _PROTOTYPE( void ack_irqs, (unsigned int) );
12304 FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
12305 FORWARD _PROTOTYPE( int w_other, (struct driver *dp, message *m_ptr) );
12306 FORWARD _PROTOTYPE( int w_hw_int, (struct driver *dp, message *m_ptr) );
12307 FORWARD _PROTOTYPE( int com_simple, (struct command *cmd) );
12308 FORWARD _PROTOTYPE( void w_timeout, (void) );
12309 FORWARD _PROTOTYPE( int w_reset, (void) );
12310 FORWARD _PROTOTYPE( void w_intr_wait, (void) );
12311 FORWARD _PROTOTYPE( int at_intr_wait, (void) );
12312 FORWARD _PROTOTYPE( int w_waitfor, (int mask, int value) );
12313 FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry) );
12314
12315 /* Entry points to this driver. */
12316 PRIVATE struct driver w_dtab = {
12317     w_name,           /* current device's name */
12318     w_do_open,         /* open or mount request, initialize device */
12319     w_do_close,        /* release device */
12320     do_ioctl,          /* get or set a partition's geometry */
12321     w_prepare,         /* prepare for I/O on a given minor device */
12322     w_transfer,        /* do the I/O */
12323     nop_cleanup,       /* nothing to clean up */
12324     w_geometry,        /* tell the geometry of the disk */
12325     nop_signal,        /* no cleanup needed on shutdown */
12326     nop_alarm,         /* ignore leftover alarms */
12327     nop_cancel,        /* ignore CANCELs */
12328     nop_select,        /* ignore selects */
12329     w_other,           /* catch-all for unrecognized commands and ioctls */
12330     w_hw_int,          /* leftover hardware interrupts */
12331 };

```

```

12332
12333 /*=====
12334 *          at_wincester_task
12335 *=====
12336 PUBLIC int main()
12337 {
12338 /* Set special disk parameters then call the generic main loop. */
12339     init_params();
12340     driver_task(&w_dtab);
12341     return(OK);
12342 }

12343 /*=====
12344 *          init_params
12345 *=====
12346 PRIVATE void init_params()
12347 {
12348 /* This routine is called at startup to initialize the drive parameters. */
12349
12350     u16_t parv[2];
12351     unsigned int vector, size;
12352     int drive, nr_drives;
12353     struct wini *wn;
12354     u8_t params[16];
12355     int s;
12356
12357 /* Boot variables. */
12358     env_parse("ata_std_timeout", "d", 0, &w_standard_timeouts, 0, 1);
12359     env_parse("ata_pci_debug", "d", 0, &w_pci_debug, 0, 1);
12360     env_parse("ata_instance", "d", 0, &w_instance, 0, 8);
12361     env_parse("ata_lba48", "d", 0, &w_lba48, 0, 1);
12362     env_parse("atapi_debug", "d", 0, &atapi_debug, 0, 1);
12363
12364     if (w_instance == 0) {
12365         /* Get the number of drives from the BIOS data area */
12366         if ((s=sys_vircopy(SELF, BIOS_SEG, NR_HD_DRIVES_ADDR,
12367                             SELF, D, (vir_bytes) params, NR_HD_DRIVES_SIZE)) != OK)
12368             panic(w_name(), "Couldn't read BIOS", s);
12369         if ((nr_drives = params[0]) > 2) nr_drives = 2;
12370
12371         for (drive = 0, wn = wini; drive < COMPAT_DRIVES; drive++, wn++) {
12372             if (drive < nr_drives) {
12373                 /* Copy the BIOS parameter vector */
12374                 vector = (drive == 0) ? BIOS_HDO_PARAMS_ADDR:BIOS_HDI_PARAMS_ADDR;
12375                 size = (drive == 0) ? BIOS_HDO_PARAMS_SIZE:BIOS_HDI_PARAMS_SIZE;
12376                 if ((s=sys_vircopy(SELF, BIOS_SEG, vector,
12377                                     SELF, D, (vir_bytes) parv, size)) != OK)
12378                     panic(w_name(), "Couldn't read BIOS", s);
12379
12380                 /* Calculate the address of the parameters and copy them */
12381                 if ((s=sys_vircopy(
12382                     SELF, BIOS_SEG, hclick_to_physb(parv[1]) + parv[0],
12383                     SELF, D, (phys_bytes) params, 16L))!=OK)
12384                     panic(w_name(),"Couldn't copy parameters", s);
12385
12386                 /* Copy the parameters to the structures of the drive */
12387                 wn->lcyliners = bp_cylinders(params);
12388                 wn->lheads = bp_heads(params);
12389                 wn->lsectors = bp_sectors(params);
12390                 wn->precomp = bp_precomp(params) >> 2;
12391             }
12392
12393             /* Fill in non-BIOS parameters. */
12394             init_drive(wn,
12395                         drive < 2 ? REG_CMD_BASE0 : REG_CMD_BASE1,
12396                         drive < 2 ? REG_CTL_BASE0 : REG_CTL_BASE1,
12397                         NO_IRQ, 0, 0, drive);

```

```

12399             w->next_drive++;
12400         }
12401     }
12402 
12403     /* Look for controllers on the pci bus. Skip none the first instance,
12404      * skip one and then 2 for every instance, for every next instance.
12405      */
12406     if (w->instance == 0)
12407         init_params_pci(0);
12408     else
12409         init_params_pci(w->instance*2-1);
12410 
12411 }

12413 #define ATA_IF_NOTCOMPAT1 (1L << 0)
12414 #define ATA_IF_NOTCOMPAT2 (1L << 2)
12415 
12416 /*-----*
12417 *           init_drive
12418 *-----*/
12419 PRIVATE void init_drive(struct wini *w int base_cmd int base_ctl int irq int ack ...
12420 {
12421     w->state = 0;
12422     w->w_status = 0;
12423     w->base_cmd = base_cmd;
12424     w->base_ctl = base_ctl;
12425     w->irq = irq;
12426     w->irq_mask = 1 << irq;
12427     w->irq_need_ack = ack;
12428     w->irq_hook_id = hook;
12429     w->lqhpfref = lqh_init(drive);
12430     w->max_count = MAX_SECS << SECTOR_SHIFT;
12431     w->lba48 = 0;
12432 }

12434 /*-----*
12435 *           init_params_pci
12436 *-----*/
12437 PRIVATE void init_params_pci(int skip)
12438 {
12439     int r, devind, drive;
12440     u16_t vid, did;
12441     pci_init();
12442     for(drive = w->next_drive; drive < MAX_DRIVES; drive++)
12443         wini[drive].state = IGNORING;
12444     for(r = pci_first_dev(&devind, &vid, &did);
12445         r!=0&&w->next_drive<MAX_DRIVES; r=pci_next_dev(&devind,&vid, &did)) {
12446         int interface, irq, irq_hook;
12447         /* Base class must be 01h (mass storage), subclass must
12448          * be 01h (ATA).
12449          */
12450         if (pci_attr_r8(devind, PCI_BCR) != 0x01 ||
12451             pci_attr_r8(devind, PCI_SCR) != 0x01) {
12452             continue;
12453         }
12454         /* Found a controller.
12455          * Programming interface register tells us more.
12456          */
12457         interface = pci_attr_r8(devind, PCI_PIFR);
12458         irq = pci_attr_r8(devind, PCI_ILR);

12459         /* Any non-compat drives? */
12460         if (interface & (ATA_IF_NOTCOMPAT1 | ATA_IF_NOTCOMPAT2)) {
12461             int s;
12462             irq_hook = irq;
12463             if (skip > 0) {
12464                 if(w->pci_debug)printf("atapci skipping contr. (remain %d)\n",skip);
12465                 skip--;
12466             }

```

```

12467             continue;
12468         }
12469         if ((s=sys_irqsetpolicy(irq, 0, &irq_hook)) != OK) {
12470             printf("atapci: couldn't set IRQ policy %d\n", irq);
12471             continue;
12472         }
12473         if ((s=sys_irqenable(&irq_hook)) != OK) {
12474             printf("atapci: couldn't enable IRQ line %d\n", irq);
12475             continue;
12476     }
12477     /* If not.. this is not the ata-pci controller we're
12478      * looking for.
12479      */
12480     if (w_pci_debug) printf("atapci skipping compatibility controller\n");
12481     continue;
12482 }
12483
12484 /* Primary channel not in compatibility mode? */
12485 if (interface & ATA_IF_NOTCOMPAT1) {
12486     u32_t base_cmd, base_ctl;
12487     base_cmd = pci_attr_r32(devind, PCI_BAR) & 0xffffffffe0;
12488     base_ctl = pci_attr_r32(devind, PCI_BAR_2) & 0xffffffffe0;
12489     if (base_cmd != REG_CMD_BASE0 && base_cmd != REG_CMD_BASE1) {
12490         init_drive(&wini[w_next_drive],
12491                     base_cmd, base_ctl, irq, 1, irq_hook, 0);
12492         init_drive(&wini[w_next_drive+1],
12493                     base_cmd, base_ctl, irq, 1, irq_hook, 1);
12494         if (w_pci_debug)
12495             printf("atapci %d: 0x%lx 0x%lx irq %d\n", devind,base_cmd,base_ctl,irq);
12496     } else printf("atapci: ignored drives on pri, base: %x\n",base_cmd);
12497 }
12498
12499 /* Secondary channel not in compatibility mode? */
12500 if (interface & ATA_IF_NOTCOMPAT2) {
12501     u32_t base_cmd, base_ctl;
12502     base_cmd = pci_attr_r32(devind, PCI_BAR_3) & 0xffffffffe0;
12503     base_ctl = pci_attr_r32(devind, PCI_BAR_4) & 0xffffffffe0;
12504     if (base_cmd != REG_CMD_BASE0 && base_cmd != REG_CMD_BASE1) {
12505         init_drive(&wini[w_next_drive+2],
12506                     base_cmd, base_ctl, irq, 1, irq_hook, 2);
12507         init_drive(&wini[w_next_drive+3],
12508                     base_cmd, base_ctl, irq, 1, irq_hook, 3);
12509         if (w_pci_debug)
12510             printf("atapci %d: 0x%lx 0x%lx irq %d\n", devind,base_cmd,base_ctl,irq);
12511     } else printf("atapci: ignored drives on secondary %x\n", base_cmd);
12512 }
12513 w_next_drive += 4;
12514 }
12515 }
12516 }

12517 /*=====
12518  *          w_do_open
12519  *=====
12520 PRIVATE int w_do_open(dp, m_ptr)
12521 struct driver *dp;
12522 message *m_ptr;
12523 {
12524 /* Device open: Initialize the controller and read the partition table. */
12525     struct wini *wn;
12526
12527     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
12528
12529     wn = w_wn;
12530
12531     /* If we've probed it before and it failed, don't probe it again. */
12532     if (wn->state & IGNORING) return ENXIO;

```

```

12535
12536 /* If we haven't identified it yet, or it's gone deaf,
12537 * (re-)identify it.
12538 */
12539 if (!(wn->state & IDENTIFIED) || (wn->state & DEAF)) {
12540     /* Try to identify the device. */
12541     if (w_identify() != OK) {
12542         if (wn->state & DEAF) w_reset();
12543         wn->state = IGNORING;
12544         return(ENXIO);
12545     }
12546     /* Do a test transaction unless it's a CD drive (then
12547     * we can believe the controller, and a test may fail
12548     * due to no CD being in the drive). If it fails, ignore
12549     * the device forever.
12550 */
12551     if (!(wn->state & ATAPI) && w_io_test() != OK) {
12552         wn->state |= IGNORING;
12553         return(ENXIO);
12554     }
12555 }
12556
12557 /* If it's not an ATAPI device, then don't open with RO_BIT. */
12558 if (!(wn->state & ATAPI) && (m_ptr->COUNT & RO_BIT)) return EACCES;
12559
12560 /* Partition the drive if it's being opened for the first time,
12561 * or being opened after being closed.
12562 */
12563 if (wn->open_ct == 0) {
12564
12565     /* Partition the disk. */
12566     memset(wn->part, sizeof(wn->part), 0);
12567     memset(wn->subpart, sizeof(wn->subpart), 0);
12568     partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY, wn->state & ATAPI);
12569 }
12570 wn->open_ct++;
12571 return(OK);
12572 }

12573 /*=====
12574 *                               w_prepare
12575 *=====
12576 */
12577 PRIVATE struct device *w_prepare(int device)
12578 {
12579 /* Prepare for I/O on a device. */
12580 struct wini *prev_wn;
12581 prev_wn = w_wn;
12582 w_device = device;
12583
12584 if (device < NR_MINORS) { /* d0, d0p[0-3], d1, ... */
12585     w_drive = device / DEV_PER_DRIVE; /* save drive number */
12586     w_wn = &wini[w_drive];
12587     w_dv = &w_wn->part[device % DEV_PER_DRIVE];
12588 } else
12589 if ((unsigned) (device - MINOR_d0p0s0) < NR_SUBDEVS) {/*d[0-7]p[0-3]s[0-3]*/
12590     w_drive = device / SUB_PER_DRIVE;
12591     w_wn = &wini[w_drive];
12592     w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
12593 } else {
12594     w_device = -1;
12595     return(NIL_DEV);
12596 }
12597 return(w_dv);
12598 }

```

```

12600 /*=====
12601 *          w_identify
12602 *=====
12603 PRIVATE int w_identify()
12604 {
12605 /* Find out if a device exists, if it is an old AT disk, or a newer ATA
12606 /* drive, a removable media device, etc.
12607 */
12608
12609     struct wini *wn = w_wn;
12610     struct command cmd;
12611     int i, s;
12612     unsigned long size;
12613 #define id_byte(n)      (&tmp_buf[2 * (n)])
12614 #define id_word(n)       (((u16_t) id_byte(n)[0] << 0) \
12615           |((u16_t) id_byte(n)[1] << 8))
12616 #define id_longword(n)  (((u32_t) id_byte(n)[0] << 0) \
12617           |((u32_t) id_byte(n)[1] << 8) \
12618           |((u32_t) id_byte(n)[2] << 16) \
12619           |((u32_t) id_byte(n)[3] << 24))
12620
12621 /* Try to identify the device. */
12622 cmd.1dh     = wn->ldhpref;
12623 cmd.command = ATA_IDENTIFY;
12624 if (com_simple(&cmd) == OK) {
12625     /* This is an ATA device. */
12626     wn->state |= SMART;
12627
12628     /* Device information. */
12629     if ((s=sys_insw(wn->base_cmd + REG_DATA, SELF, tmp_buf, SECTOR_SIZE)) != OK)
12630         panic(w_name(),"Call to sys_insw() failed", s);
12631
12632     /* Why are the strings byte swapped??? */
12633     for (i = 0; i < 40; i++) w_id_string[i] = id_byte(27)[i^1];
12634
12635     /* Preferred CHS translation mode. */
12636     wn->pcylinders = id_word(1);
12637     wn->pheads = id_word(3);
12638     wn->psectors = id_word(6);
12639     size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
12640
12641     if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
12642         /* Drive is LBA capable and is big enough to trust it to
12643          * not make a mess of it.
12644          */
12645         wn->ldhpref |= LDH_LBA;
12646         size = id_longword(60);
12647
12648         if (w_lba48 && ((id_word(83)) & (1L << 10))) {
12649             /* Drive is LBA48 capable (and LBA48 is turned on). */
12650             if (id_word(102) || id_word(103)) {
12651                 /* If no. of sectors doesn't fit in 32 bits,
12652                  * trunacte to this. So it's LBA32 for now.
12653                  * This can still address devices up to 2TB
12654                  * though.
12655                  */
12656                 size = ULONG_MAX;
12657             } else {
12658                 /* Actual number of sectors fits in 32 bits. */
12659                 size = id_longword(100);
12660             }
12661             wn->lba48 = 1;
12662         }
12663     }
12664     if (wn->lcyliners == 0) {

```

```

12667             /* No BIOS parameters? Then make some up. */
12668             wn->lcyliners = wn->pcylinders;
12669             wn->lheads = wn->pheads;
12670             wn->lsectors = wn->psectors;
12671             while (wn->lcyliners > 1024) {
12672                 wn->lheads *= 2;
12673                 wn->lcyliners /= 2;
12674             }
12675         }
12676     } else {
12677         /* Not an ATA device; no translations, no special features. Don't
12678         * touch it unless the BIOS knows about it.
12679         */
12680         if (wn->lcyliners == 0) { return(ERR); }      /* no BIOS parameters */
12681         wn->pcylinders = wn->lcyliners;
12682         wn->pheads = wn->lheads;
12683         wn->psectors = wn->lsectors;
12684         size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
12685     }
12686
12687     /* Size of the whole drive */
12688     wn->part[0].dv_size = mul64u(size, SECTOR_SIZE);
12689
12690     /* Reset/calibrate (where necessary) */
12691     if (w_specify() != OK && w_specify() != OK) {
12692         return(ERR);
12693     }
12694
12695     if (wn->irq == NO_IRQ) {
12696         /* Everything looks OK; register IRQ so we can stop polling. */
12697         wn->irq = w_drive < 2 ? AT_WINI_0_IRQ : AT_WINI_1_IRQ;
12698         wn->irq_hook_id = wn->irq; /* id to be returned if interrupt occurs */
12699         if ((s=sys_irqsetpolicy(wn->irq, IRQ_REENABLE, &wn->irq_hook_id)) != OK)
12700             panic(w_name(), "couldn't set IRQ policy", s);
12701         if ((s=sys_irqenable(&wn->irq_hook_id)) != OK)
12702             panic(w_name(), "couldn't enable IRQ line", s);
12703     }
12704     wn->state |= IDENTIFIED;
12705     return(OK);
12706 }
12707
12708 /*=====
12709 *           w_name
12710 *=====
12711 PRIVATE char *w_name()
12712 {
12713     /* Return a name for the current device. */
12714     static char name[] = "AT-D0";
12715
12716     name[4] = '0' + w_drive;
12717     return name;
12718 }
12719
12720 /*=====
12721 *           w_io_test
12722 *=====
12723 PRIVATE int w_io_test(void)
12724 {
12725     int r, save_dev;
12726     int save_timeout, save_errors, save_wakeup;
12727     iovec_t iov;
12728     static char buf[SECTOR_SIZE];
12729     iov.iov_addr = (vir_bytes) buf;
12730     iov.iov_size = sizeof(buf);
12731     save_dev = w_device;
12732
12733     /* Reduce timeout values for this test transaction. */

```

```

12734     save_timeout = timeout_ticks;
12735     save_errors = max_errors;
12736     save_wakeup = wakeup_ticks;
12737
12738     if (!w_standard_timeouts) {
12739         timeout_ticks = HZ * 4;
12740         wakeup_ticks = HZ * 6;
12741         max_errors = 3;
12742     }
12743
12744     w_testing = 1;
12745
12746     /* Try I/O on the actual drive (not any (sub)partition). */
12747     if (w_prepare(w_drive + DEV_PER_DRIVE) == NIL_DEV)
12748         panic(w_name(), "Couldn't switch devices", NO_NUM);
12749
12750     r = w_transfer(SELF, DEV_GATHER, 0, &iov, 1);
12751
12752     /* Switch back. */
12753     if (w_prepare(save_dev) == NIL_DEV)
12754         panic(w_name(), "Couldn't switch back devices", NO_NUM);
12755
12756     /* Restore parameters. */
12757     timeout_ticks = save_timeout;
12758     max_errors = save_errors;
12759     wakeup_ticks = save_wakeup;
12760     w_testing = 0;
12761
12762     /* Test if everything worked. */
12763     if (r != OK || iov.iov_size != 0) {
12764         return ERR;
12765     }
12766
12767     /* Everything worked. */
12768
12769     return OK;
12770 }
12771 /*=====
12772 *          w_specify
12773 *=====
12774 PRIVATE int w_specify()
12775 {
12776     /* Routine to initialize the drive after boot or when a reset is needed. */
12777
12778     struct wini *wn = w_wn;
12779     struct command cmd;
12780
12781     if ((wn->state & DEAF) && w_reset() != OK) {
12782         return(ERR);
12783     }
12784
12785     if (!(wn->state & ATAPI)) {
12786         /* Specify parameters: precompensation, number of heads and sectors. */
12787         cmd.precomp = wn->precomp;
12788         cmd.count   = wn->psectors;
12789         cmd.ldh    = w_wn->lhpref | (wn->pheads - 1);
12790         cmd.command = CMD_SPECIFY;           /* Specify some parameters */
12791
12792         /* Output command block and see if controller accepts the parameters. */
12793         if (com_simple(&cmd) != OK) return(ERR);
12794
12795         if (!(wn->state & SMART)) {
12796             /* Calibrate an old disk. */
12797             cmd.sector  = 0;
12798             cmd.cyl_lo = 0;
12799             cmd.cyl_hi = 0;
12800             cmd.ldh     = w_wn->lhpref;
12801

```

```

12802                 cmd.command = CMD_RECALIBRATE;
12803
12804         if (com_simple(&cmd) != OK) return(ERR);
12805     }
12806 }
12807 wn->state |= INITIALIZED;
12808 return(OK);
12809 }

12811 /*=====
12812 *          do_transfer
12813 *=====
12814 PRIVATE int do_transfer(struct wini *wn, unsigned int precomp, unsigned int count,
12815             unsigned int sector, unsigned int opcode)
12816 {
12817     struct command cmd;
12818     unsigned secspcyl = wn->pheads * wn->psectors;
12819
12820     cmd.precomp = precomp;
12821     cmd.count = count;
12822     cmd.command = opcode == DEV_SCATTER ? CMD_WRITE : CMD_READ;
12823     /*
12824     if (w_lba48 && wn->lba48) {
12825     } else */
12826     if (wn->ldhpref & LDH_LBA) {
12827         cmd.sector = (sector >> 0) & 0xFF;
12828         cmd.cyl_lo = (sector >> 8) & 0xFF;
12829         cmd.cyl_hi = (sector >> 16) & 0xFF;
12830         cmd.1dh = wn->ldhpref | ((sector >> 24) & 0xF);
12831     } else {
12832         int cylinder, head, sec;
12833         cylinder = sector / secspcyl;
12834         head = (sector % secspcyl) / wn->psectors;
12835         sec = sector % wn->psectors;
12836         cmd.sector = sec + 1;
12837         cmd.cyl_lo = cylinder & BYTE;
12838         cmd.cyl_hi = (cylinder >> 8) & BYTE;
12839         cmd.1dh = wn->ldhpref | head;
12840     }
12841
12842     return com_out(&cmd);
12843 }

12845 /*=====
12846 *          w_transfer
12847 *=====
12848 PRIVATE int w_transfer(proc_nr, opcode, position, iov, nr_req)
12849 int proc_nr;           /* process doing the request */
12850 int opcode;            /* DEV_GATHER or DEV_SCATTER */
12851 off_t position;       /* offset on device to read or write */
12852 iovec_t *iov;          /* pointer to read or write request vector */
12853 unsigned nr_req;       /* length of request vector */
12854 {
12855     struct wini *wn = w_wn;
12856     iovec_t *iop, *iov_end = iov + nr_req;
12857     int r, s, errors;
12858     unsigned long block;
12859     unsigned long dv_size = cv64ul(w_dv->dv_size);
12860     unsigned cylinder, head, sector, nbytes;
12861
12862     /* Check disk address. */
12863     if ((position & SECTOR_MASK) != 0) return(EINVAL);
12864
12865     errors = 0;
12866
12867     while (nr_req > 0) {
12868         /* How many bytes to transfer? */

```

```

12869     nbytes = 0;
12870     for (iop = iov; iop < iov_end; iop++) nbytes += iop->iov_size;
12871     if ((nbytes & SECTOR_MASK) != 0) return(EINVAL);
12872
12873     /* Which block on disk and how close to EOF? */
12874     if (position >= dv_size) return(OK); /* At EOF */
12875     if (position + nbytes > dv_size) nbytes = dv_size - position;
12876     block = div64u(add64u(w_dv->dv_base, position), SECTOR_SIZE);
12877
12878     if (nbytes >= wn->max_count) {
12879         /* The drive can't do more than max_count at once. */
12880         nbytes = wn->max_count;
12881     }
12882
12883     /* First check to see if a reinitialization is needed. */
12884     if (!(wn->state & INITIALIZED) && w_specify() != OK) return(EIO);
12885
12886     /* Tell the controller to transfer nbytes bytes. */
12887     r = do_transfer(wn, wn->precomp, ((nbytes >> SECTOR_SHIFT) & BYTE),
12888                     block, opcode);
12889
12890     while (r == OK && nbytes > 0) {
12891         /* For each sector, wait for an interrupt and fetch the data
12892         * (read), or supply data to the controller and wait for an
12893         * interrupt (write).
12894         */
12895
12896         if (opcode == DEV_GATHER) {
12897             /* First an interrupt, then data. */
12898             if ((r = at_intr_wait()) != OK) {
12899                 /* An error, send data to the bit bucket. */
12900                 if (w_wn->w_status & STATUS_DRQ) {
12901                     if ((s=sys_insw(wn->base_cmd + REG_DATA, SELF, tmp_buf, SECTOR_SIZE)) != OK)
12902                         panic(w_name(),"Call to sys_insw() failed", s);
12903                     }
12904                     break;
12905                 }
12906             }
12907
12908             /* Wait for data transfer requested. */
12909             if (!w_waitfor(STATUS_DRQ, STATUS_DRQ)) { r = ERR; break; }
12910
12911             /* Copy bytes to or from the device's buffer. */
12912             if (opcode == DEV_GATHER) { if((s=sys_insw(wn->base_cmd+REG_DATA,
12913                                         proc_nr,(void*)iov->iov_addr,SECTOR_SIZE))!=OK)
12914                 panic(w_name(),"Call to sys_insw() failed", s);
12915             } else { if((s=sys_outsw(wn->base_cmd+REG_DATA,proc_nr,
12916                                         (void *) iov->iov_addr,SECTOR_SIZE))!=OK)
12917                 panic(w_name(),"Call to sys_insw() failed", s);
12918
12919                 /* Data sent, wait for an interrupt. */
12920                 if ((r = at_intr_wait()) != OK) break;
12921             }
12922
12923             /* Book the bytes successfully transferred. */
12924             nbytes -= SECTOR_SIZE;
12925             position += SECTOR_SIZE;
12926             iov->iov_addr += SECTOR_SIZE;
12927             if ((iov->iov_size -= SECTOR_SIZE) == 0) { iov++; nr_req--; }
12928         }
12929
12930         /* Any errors? */
12931         if (r != OK) {
12932             /* Don't retry if sector marked bad or too many errors. */
12933             if (r == ERR_BAD_SECTOR || ++errors == max_errors) {
12934                 w_command = CMD_IDLE;
12935                 return(EIO);
12936             }

```

```

12937      }
12938  }
12939
12940  w_command = CMD_IDLE;
12941  return(OK);
12942 }

12944 /*=====
12945 *          com_out
12946 *=====
12947 PRIVATE int com_out(cmd)
12948 struct command *cmd;           /* Command block */
12949 {
12950 /* Output the command block to the winchester controller and return status */
12951
12952     struct wini *wn = w_wn;
12953     unsigned base_cmd = wn->base_cmd;
12954     unsigned base_ctl = wn->base_ctl;
12955     pvb_pair_t outbyte[7];           /* vector for sys_voutb() */
12956     int s;                         /* status for sys_(v)outb() */
12957
12958     if (w_wn->state & IGNORING) return ERR;
12959
12960     if (!w_waitfor(STATUS_BSY, 0)) {
12961         printf("%s: controller not ready\n", w_name());
12962         return(ERR);
12963     }
12964
12965     /* Select drive. */
12966     if ((s=sys_outb(base_cmd + REG_LDH, cmd->ldh)) != OK)
12967         panic(w_name(),"Couldn't write register to select drive",s);
12968
12969     if (!w_waitfor(STATUS_BSY, 0)) {
12970         printf("%s: com_out: drive not ready\n", w_name());
12971         return(ERR);
12972     }
12973
12974 /* Schedule a wakeup call, some controllers are flaky. This is done with
12975 * a synchronous alarm. If a timeout occurs a SYN_ALARM message is sent
12976 * from HARDWARE, so that w_intr_wait() can call w_timeout() in case the
12977 * controller was not able to execute the command. Leftover timeouts are
12978 * simply ignored by the main loop.
12979 */
12980 sys_setalarm(wakeup_ticks, 0);
12981
12982     wn->w_status = STATUS_ADMBSY;
12983     w_command = cmd->command;
12984     pv_set(outbyte[0], base_ctl + REG_CTL, wn->phheads >= 8 ? CTL_EIGHTHEADS : 0);
12985     pv_set(outbyte[1], base_cmd + REG_PRECOMP, cmd->precomp);
12986     pv_set(outbyte[2], base_cmd + REG_COUNT, cmd->count);
12987     pv_set(outbyte[3], base_cmd + REG_SECTOR, cmd->sector);
12988     pv_set(outbyte[4], base_cmd + REG_CYL_LO, cmd->cyl_lo);
12989     pv_set(outbyte[5], base_cmd + REG_CYL_HI, cmd->cyl_hi);
12990     pv_set(outbyte[6], base_cmd + REG_COMMAND, cmd->command);
12991     if ((s=sys_voutb(outbyte,7)) != OK)
12992         panic(w_name(),"Couldn't write registers with sys_voutb()",s);
12993     return(OK);
12994 }

12996 /*=====
12997 *          w_need_reset
12998 *=====
12999 PRIVATE void w_need_reset()
13000 {
13001 /* The controller needs to be reset. */
13002     struct wini *wn;
13003     int dr = 0;
13004

```

```

13005    for (wn = wini; wn < &wini[MAX_DRIVES]; wn++, dr++) {
13006        if (wn->base_cmd == w_wn->base_cmd) {
13007            wn->state |= DEAF;
13008            wn->state &= ~INITIALIZED;
13009        }
13010    }
13011 }

13013 /*=====
13014 *          w_do_close
13015 *=====
13016 PRIVATE int w_do_close(dp, m_ptr)
13017 struct driver *dp;
13018 message *m_ptr;
13019 {
13020 /* Device close: Release a device. */
13021     if (w_prepare(m_ptr->DEVICE) == NIL_DEV)
13022         return(ENXIO);
13023     w_wn->open_ct--;
13024     return(OK);
13025 }

13027 /*=====
13028 *          com_simple
13029 *=====
13030 PRIVATE int com_simple(cmd)
13031 struct command *cmd;           /* Command block */
13032 {
13033 /* A simple controller command, only one interrupt and no data-out phase. */
13034     int r;
13035
13036     if (w_wn->state & IGNORING) return ERR;
13037
13038     if ((r = com_out(cmd)) == OK) r = at_intr_wait();
13039     w_command = CMD_IDLE;
13040     return(r);
13041 }

13043 /*=====
13044 *          w_timeout
13045 *=====
13046 PRIVATE void w_timeout(void)
13047 {
13048     struct wini *wn = w_wn;
13049
13050     switch (w_command) {
13051     case CMD_IDLE:
13052         break;           /* fine */
13053     case CMD_READ:
13054     case CMD_WRITE:
13055         /* Impossible, but not on PC's: The controller does not respond. */
13056
13057         /* Limiting multisector I/O seems to help. */
13058         if (wn->max_count > 8 * SECTOR_SIZE) {
13059             wn->max_count = 8 * SECTOR_SIZE;
13060         } else {
13061             wn->max_count = SECTOR_SIZE;
13062         }
13063         /*FALL THROUGH*/
13064     default:
13065         /* Some other command. */
13066         if (w_testing) wn->state |= IGNORING; /* Kick out this drive. */
13067         else if (!w_silent) printf("%s: timeout on command %02x\n", w_name(), w_command);
13068         w_need_reset();
13069         wn->w_status = 0;
13070     }
13071 }

```

```

13073 /*=====
13074 *          w_reset
13075 *=====
13076 PRIVATE int w_reset()
13077 {
13078 /* Issue a reset to the controller. This is done after any catastrophe,
13079 * like the controller refusing to respond.
13080 */
13081     int s;
13082     struct wini *wn = w_wn;
13083
13084 /* Don't bother if this drive is forgotten. */
13085 if (w_wn->state & IGNORING) return ERR;
13086
13087 /* Wait for any internal drive recovery. */
13088 tickdelay(RECOVERY_TICKS);
13089
13090 /* Strobe reset bit */
13091 if ((s=sys_outb(wn->base_ctl + REG_CTL, CTL_RESET)) != OK)
13092     panic(w_name(),"Couldn't strobe reset bit",s);
13093 tickdelay(DELAY_TICKS);
13094 if ((s=sys_outb(wn->base_ctl + REG_CTL, 0)) != OK)
13095     panic(w_name(),"Couldn't strobe reset bit",s);
13096 tickdelay(DELAY_TICKS);
13097
13098 /* Wait for controller ready */
13099 if (!w_waitfor(STATUS_BSY, 0)) {
13100     printf("%s: reset failed, drive busy\n", w_name());
13101     return(ERR);
13102 }
13103
13104 /* The error register should be checked now, but some drives mess it up. */
13105
13106 for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
13107     if (wn->base_cmd == w_wn->base_cmd) {
13108         wn->state &= ~DEAF;
13109         if (w_wn->irq_need_ack) {
13110             /* Make sure irq is actually enabled.. */
13111             sys_irqenable(&w_wn->irq_hook_id);
13112         }
13113     }
13114 }
13115
13116
13117     return(OK);
13118 }

13120 /*=====
13121 *          w_intr_wait
13122 *=====
13123 PRIVATE void w_intr_wait()
13124 {
13125 /* Wait for a task completion interrupt. */
13126
13127     message m;
13128
13129     if (w_wn->irq != NO_IRQ) {
13130         /* Wait for an interrupt that sets w_status to "not busy". */
13131         while (w_wn->w_status & (STATUS_ADMBSY|STATUS_BSY)) {
13132             receive(ANY, &m);           /* expect HARD_INT message */
13133             if (m.m_type == SYN_ALARM) { /* but check for timeout */
13134                 w_timeout();
13135             } else if (m.m_type == HARD_INT) {
13136                 sys_inb(w_wn->base_cmd + REG_STATUS, &w_wn->w_status);
13137                 ack_irqs(m.NOTIFY_ARG);
13138             } else {
13139                 printf("AT_WINI got unexpected message %d from %d\n",
13140                         m.m_type, m.m_source);

```

```

13141         }
13142     }
13143 } else {
13144     /* Interrupt not yet allocated; use polling. */
13145     (void) w_waitfor(STATUS_BSY, 0);
13146 }
13147 }

13149 /*=====
13150 *          at_intr_wait
13151 *=====
13152 PRIVATE int at_intr_wait()
13153 {
13154     /* Wait for an interrupt, study the status bits and return error/success. */
13155     int r;
13156     int s,inval;      /* read value with sys_inb */
13157
13158     w_intr_wait();
13159     if ((w_wn->w_status & (STATUS_BSY | STATUS_WF | STATUS_ERR)) == 0) {
13160         r = OK;
13161     } else {
13162         if ((s=sys_inb(w_wn->base_cmd + REG_ERROR, &inval)) != OK)
13163             panic(w_name(),"Couldn't read register",s);
13164         if ((w_wn->w_status & STATUS_ERR) && (inval & ERROR_BB)) {
13165             r = ERR_BAD_SECTOR;      /* sector marked bad, retries won't help */
13166         } else {
13167             r = ERR;                  /* any other error */
13168         }
13169     }
13170     w_wn->w_status |= STATUS_ADMBSY;      /* assume still busy with I/O */
13171     return(r);
13172 }

13174 /*=====
13175 *          w_waitfor
13176 *=====
13177 PRIVATE int w_waitfor(mask, value)
13178 int mask;           /* status mask */
13179 int value;          /* required status */
13180 {
13181     /* Wait until controller is in the required state. Return zero on timeout.
13182     * An alarm that set a timeout flag is used. TIMEOUT is in micros, we need
13183     * ticks. Disabling the alarm is not needed, because a static flag is used
13184     * and a leftover timeout cannot do any harm.
13185     */
13186     clock_t t0, t1;
13187     int s;
13188     getuptime(&t0);
13189     do {
13190         if ((s=sys_inb(w_wn->base_cmd + REG_STATUS, &w_wn->w_status)) != OK).
13191             panic(w_name(),"Couldn't read register",s);
13192         if ((w_wn->w_status & mask) == value) {
13193             return 1;
13194         }
13195     } while ((s=getuptime(&t1)) == OK && (t1-t0) < timeout_ticks );
13196     if (OK != s) printf("AT_WINI: warning, get_uptime failed: %d\n",s);
13197
13198     w_need_reset();          /* controller gone deaf */
13199     return(0);
13200 }

13202 /*=====
13203 *          w_geometry
13204 *=====
13205 PRIVATE void w_geometry(entry)
13206 struct partition *entry;
13207 {

```

```

13208     struct wini *wn = w_wn;
13209
13210     if (wn->state & ATAPI) {           /* Make up some numbers. */
13211         entry->cylinders = div64u(wn->part[0].dv_size, SECTOR_SIZE) / (64*32);
13212         entry->heads = 64;
13213         entry->sectors = 32;
13214     } else {                           /* Return logical geometry. */
13215         entry->cylinders = wn->lcyldinders;
13216         entry->heads = wn->lheads;
13217         entry->sectors = wn->lsectors;
13218     }
13219 }
13220
13221 /*-----*
13222 *          w_other
13223 *-----*/
13224 PRIVATE int w_other(dr, m)
13225 struct driver *dr;
13226 message *m;
13227 {
13228     int r, timeout, prev;
13229
13230     if (m->m_type != DEV_IOCTL) {
13231         return EINVAL;
13232     }
13233
13234     if (m->REQUEST == DIOCTIMEOUT) {
13235         if ((r=sys_datacopy(m->PROC_NR, (vir_bytes)m->ADDRESS,
13236                         SELF, (vir_bytes)&timeout, sizeof(timeout))) != OK)
13237             return r;
13238
13239         if (timeout == 0) {
13240             /* Restore defaults. */
13241             timeout_ticks = DEF_TIMEOUT_TICKS;
13242             max_errors = MAX_ERRORS;
13243             wakeup_ticks = WAKEUP;
13244             w_silent = 0;
13245         } else if (timeout < 0) {
13246             return EINVAL;
13247         } else {
13248             prev = wakeup_ticks;
13249
13250             if (!w_standard_timeouts) {
13251                 /* Set (lower) timeout, lower error
13252                  tolerance and set silent mode.
13253                  */
13254                 wakeup_ticks = timeout;
13255                 max_errors = 3;
13256                 w_silent = 1;
13257
13258                 if (timeout_ticks > timeout)
13259                     timeout_ticks = timeout;
13260             }
13261
13262             if ((r=sys_datacopy(SELF, (vir_bytes)&prev,
13263                               m->PROC_NR,(vir_bytes)m->ADDRESS,sizeof(prev)))!=OK)
13264                 return r;
13265         }
13266
13267     } else if (m->REQUEST == DIOCOPENCT) {
13268         int count;
13269         if (w_prepare(m->DEVICE) == NIL_DEV) return ENXIO;
13270         count = w_wn->open_ct;
13271         if ((r=sys_datacopy(SELF, (vir_bytes)&count,
13272                           m->PROC_NR, (vir_bytes)m->ADDRESS, sizeof(count))) != OK)
13273             return r;
13274
13275     return OK;

```

```

13276      }
13277      return EINVAL;
13278  }

13280 /*=====
13281 *          w_hw_int
13282 *=====
13283 PRIVATE int w_hw_int(dr, m)
13284 struct driver *dr;
13285 message *m;
13286 {
13287 /* Leftover interrupt(s) received; ack it/them. */
13288 ack_irqs(m->NOTIFY_ARG);
13289
13290      return OK;
13291 }

13294 /*=====
13295 *          ack_irqs
13296 *=====
13297 PRIVATE void ack_irqs(unsigned int irqs)
13298 {
13299     unsigned int drive;
13300     for (drive = 0; drive < MAX_DRIVES && irqs; drive++) {
13301         if (!(wini[drive].state & IGNORING) && wini[drive].irq_need_ack &&
13302             (wini[drive].irq_mask & irqs)) {
13303             if (sys_inb((wini[drive].base_cmd+REG_STATUS),&wini[drive].w_status)!=OK)
13304                 printf("couldn't ack irq on drive %d\n", drive);
13305             if (sys_irqenable(&wini[drive].irq_hook_id) != OK)
13306                 printf("couldn't re-enable drive %d\n", drive);
13307             irqs &= ~wini[drive].irq_mask;
13308         }
13309     }
13310 }

13313 #define STSTR(a) if (status & STATUS_ ## a) { strcat(str, #a); strcat(str, " ");}
13314 #define ERRSTR(a) if (e & ERROR_ ## a) { strcat(str, #a); strcat(str, " ");}
13315 char *strstatus(int status)
13316 {
13317     static char str[200];
13318     str[0] = '\0';
13319
13320     STSTR(BSY);
13321     STSTR(DRDY);
13322     STSTR(DMADF);
13323     STSTR(SRVCDS);
13324     STSTR(DRQ);
13325     STSTR(CORR);
13326     STSTR(CHECK);
13327     return str;
13328 }

13330 char *strerr(int e)
13331 {
13332     static char str[200];
13333     str[0] = '\0';
13334
13335     ERRSTR(BB);
13336     ERRSTR(ECC);
13337     ERRSTR(ID);
13338     ERRSTR(AC);
13339     ERRSTR(TK);
13340     ERRSTR(DM);
13341
13342     return str;
13343 }

```

```
#####
drivers/tty/tty.h
#####

13400 /*      tty.h - Terminals      */
13401
13402 #include <timers.h>
13403
13404 /* First minor numbers for the various classes of TTY devices. */
13405 #define CONS_MINOR          0
13406 #define LOG_MINOR           15
13407 #define RS232_MINOR         16
13408 #define TTYPX_MINOR         128
13409 #define PTYPX_MINOR         192
13410
13411 #define LINERWAP            1 /* console.c - wrap lines at column 80 */
13412
13413 #define TTY_IN_BYTES        256 /* tty input queue size */
13414 #define TAB_SIZE              8 /* distance between tab stops */
13415 #define TAB_MASK             7 /* mask to compute a tab stop position */
13416
13417 #define ESC                 '\33' /* escape */
13418
13419 #define O_NOCTTY            00400 /* from <fcntl.h>, or cc will choke */
13420 #define O_NONBLOCK           04000
13421
13422 struct tty;
13423 typedef _PROTOTYPE( int (*devfun_t), (struct tty *tp, int try_only) );
13424 typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );
13425
13426 typedef struct tty {
13427     int tty_events;           /* set when TTY should inspect this line */
13428     int tty_index;            /* index into TTY table */
13429     int tty_minor;            /* device minor number */
13430
13431     /* Input queue. Typed characters are stored here until read by a program. */
13432     u16_t *tty_inhead;        /* pointer to place where next char goes */
13433     u16_t *tty_intail;        /* pointer to next char to be given to prog */
13434     int tty_incount;         /* # chars in the input queue */
13435     int tty_eotct;            /* number of "line breaks" in input queue */
13436     devfun_t tty_devread;    /* routine to read from low level buffers */
13437     devfun_t tty_icancel;    /* cancel any device input */
13438     int tty_min;              /* minimum requested #chars in input queue */
13439     timer_t tty_tmr;         /* the timer for this tty */
13440
13441     /* Output section. */
13442     devfun_t tty_devwrite;   /* routine to start actual device output */
13443     devfunarg_t tty_echo;    /* routine to echo characters input */
13444     devfun_t tty_ocancel;    /* cancel any ongoing device output */
13445     devfun_t tty_break;      /* let the device send a break */
13446
13447     /* Terminal parameters and status. */
13448     int tty_position;         /* current position on the screen for echoing */
13449     char tty_reprint;         /* 1 when echoed input messed up, else 0 */
13450     char tty_escaped;         /* 1 when LNEXT (^V) just seen, else 0 */
13451     char tty_inhibited;      /* 1 when STOP (^S) just seen (stops output) */
13452     char tty_pgrp;            /* slot number of controlling process */
13453     char tty_openct;          /* count of number of opens of this tty */
13454
13455     /* Information about incomplete I/O requests is stored here. */
13456     char tty_inrepcode;       /* reply code, TASK_REPLY or REVIVE */
13457     char tty_inrevived;       /* set to 1 if revive callback is pending */
13458     char tty_incaller;        /* process that made the call (usually FS) */
13459     char tty_inproc;          /* process that wants to read from tty */
13460     vir_bytes tty_in_vir;     /* virtual address where data is to go */
13461     int tty_inleft;           /* how many chars are still needed */
13462     int tty_incum;            /* # chars input so far */

```

```

13463     char tty_outrepcode;          /* reply code, TASK_REPLY or REVIVE */
13464     char tty_outrevived;         /* set to 1 if revive callback is pending */
13465     char tty_outcaller;          /* process that made the call (usually FS) */
13466     char tty_outproc;           /* process that wants to write to tty */
13467     vir_bytes tty_out_vir;       /* virtual address where data comes from */
13468     int tty_outleft;            /* # chars yet to be output */
13469     int tty_outcum;             /* # chars output so far */
13470     char tty_iocaller;          /* process that made the call (usually FS) */
13471     char tty_ioproc;            /* process that wants to do an ioctl */
13472     int tty_ioreq;              /* ioctl request code */
13473     vir_bytes tty_iovir;        /* virtual address of ioctl buffer */

13474
13475     /* select() data */
13476     int tty_select_ops;          /* which operations are interesting */
13477     int tty_select_proc;         /* which process wants notification */

13478
13479     /* Miscellaneous. */
13480     devfun_t tty_ioctl;          /* set line speed, etc. at the device level */
13481     devfun_t tty_close;          /* tell the device that the tty is closed */
13482     void *tty_priv;              /* pointer to per device private data */
13483     struct termios tty_termios;   /* terminal attributes */
13484     struct winsize tty_winsize;   /* window size (#lines and #columns) */

13485
13486     u16_t tty_inbuf[TTY_IN_BYTES];/* tty input buffer */

13487
13488 } tty_t;

13489
13490 /* Memory allocated in tty.c, so extern here. */
13491 extern tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
13492 extern int ccurrent;           /* currently visible console */
13493 extern int irq_hook_id;        /* hook id for keyboard irq */

13494
13495 extern unsigned long kbd_irq_set;
13496 extern unsigned long rs_irq_set;
13497
13498 /* Values for the fields. */
13499 #define NOT_ESCAPED      0 /* previous character is not LNEXT (^V) */
13500 #define ESCAPED          1 /* previous character was LNEXT (^V) */
13501 #define RUNNING          0 /* no STOP (^S) has been typed to stop output */
13502 #define STOPPED          1 /* STOP (^S) has been typed to stop output */

13503
13504 /* Fields and flags on characters in the input queue. */
13505 #define IN_CHAR          0x00FF /* low 8 bits are the character itself */
13506 #define IN_LEN            0x0F00 /* length of char if it has been echoed */
13507 #define IN_LSHIFT          8 /* length = (c & IN_LEN) >> IN_LSHIFT */
13508 #define IN_EOT             0x1000 /* char is a line break (^D, LF) */
13509 #define IN_EOF             0x2000 /* char is EOF (^D), do not return to user */
13510 #define IN_ESC              0x4000 /* escaped by LNEXT (^V), no interpretation */

13511
13512 /* Times and timeouts. */
13513 #define force_timeout() ((void) (0))

13514
13515 /* Memory allocated in tty.c, so extern here. */
13516 extern timer_t *tty_timers;    /* queue of TTY timers */
13517 extern clock_t tty_next_timeout; /* next TTY timeout */

13518
13519 /* Number of elements and limit of a buffer. */
13520 #define buflen(buf)      (sizeof(buf) / sizeof((buf)[0]))
13521 #define bufend(buf)       ((buf) + buflen(buf))

13522
13523 /* Memory allocated in tty.c, so extern here. */
13524 extern struct machine machine; /* machine information (a.o.: pc_at, ega) */

13525
13526 /* Function prototypes for TTY driver. */
13527 /* tty.c */
13528 _PROTOTYPE( void handle_events, (struct tty *tp) );;
13529 _PROTOTYPE( void sigchar, (struct tty *tp, int sig) );

```

```

13530 _PROTOTYPE( void tty_task, (void) );
13531 _PROTOTYPE( int in_process, (struct tty *tp, char *buf, int count) );
13532 _PROTOTYPE( void out_process, (struct tty *tp, char *bstart, char *bpos,
13533                                char *bend, int *icount, int *ocount) );
13534 _PROTOTYPE( void tty_wakeup, (clock_t now) );
13535 _PROTOTYPE( void tty_reply, (int code, int replyee, int proc_nr,
13536                                int status) );
13537 _PROTOTYPE( int tty_devnop, (struct tty *tp, int try) );
13538 _PROTOTYPE( int select_try, (struct tty *tp, int ops) );
13539 _PROTOTYPE( int select_retry, (struct tty *tp) );
13540
13541 /* console.c */
13542 _PROTOTYPE( void kputc, (int c) );
13543 _PROTOTYPE( void cons_stop, (void) );
13544 _PROTOTYPE( void do_new_kmess, (message *m) );
13545 _PROTOTYPE( void do_diagnostics, (message *m) );
13546 _PROTOTYPE( void scr_init, (struct tty *tp) );
13547 _PROTOTYPE( void toggle_scroll, (void) );
13548 _PROTOTYPE( int con_loadfont, (message *m) );
13549 _PROTOTYPE( void select_console, (int cons_line) );
13550
13551 /* keyboard.c */
13552 _PROTOTYPE( void kb_init, (struct tty *tp) );
13553 _PROTOTYPE( void kb_init_once, (void) );
13554 _PROTOTYPE( int kbd_loadmap, (message *m) );
13555 _PROTOTYPE( void do_panic.dumps, (message *m) );
13556 _PROTOTYPE( void do_fkey_ctl, (message *m) );
13557 _PROTOTYPE( void kbd_interrupt, (message *m) );
13558
13559 /* vidcopy.s */
13560 _PROTOTYPE( void vid_vid_copy, (unsigned src, unsigned dst, unsigned count));
13561 _PROTOTYPE( void mem_vid_copy, (u16_t *src, unsigned dst, unsigned count));

```

+++++  
drivers/tty/tty.c  
+++++

```

13600 /* This file contains the terminal driver, both for the IBM console and regular
13601 * ASCII terminals. It handles only the device-independent part of a TTY, the
13602 * device dependent parts are in console.c, rs232.c, etc. This file contains
13603 * two main entry points, tty_task() and tty_wakeup(), and several minor entry
13604 * points for use by the device-dependent code.
13605 *
13606 * The device-independent part accepts "keyboard" input from the device-
13607 * dependent part, performs input processing (special key interpretation),
13608 * and sends the input to a process reading from the TTY. Output to a TTY
13609 * is sent to the device-dependent code for output processing and "screen"
13610 * display. Input processing is done by the device by calling 'in_process'
13611 * on the input characters, output processing may be done by the device itself
13612 * or by calling 'out_process'. The TTY takes care of input queuing, the
13613 * device does the output queuing. If a device receives an external signal,
13614 * like an interrupt, then it causes tty_wakeup() to be run by the CLOCK task
13615 * to, you guessed it, wake up the TTY to check if input or output can
13616 * continue.
13617 *
13618 * The valid messages and their parameters are:
13619 *
13620 * HARD_INT:      output has been completed or input has arrived
13621 * SYS_SIG:       e.g., MINIX wants to shutdown; run code to cleanly stop
13622 * DEV_READ:      a process wants to read from a terminal
13623 * DEV_WRITE:     a process wants to write on a terminal
13624 * DEV_IOCTL:    a process wants to change a terminal's parameters
13625 * DEV_OPEN:      a tty line has been opened

```

```

13626 *  DEV_CLOSE:      a tty line has been closed
13627 *  DEV_SELECT:    start select notification request
13628 *  DEV_STATUS:    FS wants to know status for SELECT or REVIVE
13629 *  CANCEL:        terminate a previous incomplete system call immediately
13630 *
13631 *      m_type      TTY_LINE   PROC_NR   COUNT   TTY_SPEK   TTY_FLAGS   ADDRESS
13632 *
13633 * | HARD_INT | | | | | | | |
13634 * |-----+-----+-----+-----+-----+-----+
13635 * | SYS_SIG | sig set | | | | |
13636 * |-----+-----+-----+-----+-----+-----+
13637 * | DEV_READ |minor dev| proc nr | count | 0_NONBLOCK| buf ptr
13638 * |-----+-----+-----+-----+-----+-----+
13639 * | DEV_WRITE |minor dev| proc nr | count | | buf ptr
13640 * |-----+-----+-----+-----+-----+-----+
13641 * | DEV_IOCTL |minor dev| proc nr | func code|erase etc| flags
13642 * |-----+-----+-----+-----+-----+-----+
13643 * | DEV_OPEN  |minor dev| proc nr | 0_NOCTTY| |
13644 * |-----+-----+-----+-----+-----+-----+
13645 * | DEV_CLOSE |minor dev| proc nr | | | |
13646 * |-----+-----+-----+-----+-----+-----+
13647 * | DEV_STATUS | | | | | |
13648 * |-----+-----+-----+-----+-----+-----+
13649 * | CANCEL    |minor dev| proc nr | | | |
13650 *
13651 *
13652 * Changes:
13653 *   Jan 20, 2004   moved TTY driver to user-space (Jorrit N. Herder)
13654 *   Sep 20, 2004   local timer management/ sync alarms (Jorrit N. Herder)
13655 *   Jul 13, 2004   support for function key observers (Jorrit N. Herder)
13656 */
13657
13658 #include "../drivers.h"
13659 #include "../drivers.h"
13660 #include <termios.h>
13661 #include <sys/ioc_tty.h>
13662 #include <signal.h>
13663 #include <minix/callnr.h>
13664 #include <minix/keymap.h>
13665 #include "tty.h"
13666
13667 #include <sys/time.h>
13668 #include <sys/select.h>
13669
13670 extern int irq_hook_id;
13671
13672 unsigned long kbd_irq_set = 0;
13673 unsigned long rs_irq_set = 0;
13674
13675 /* Address of a tty structure. */
13676 #define tty_addr(line) (&tty_table[line])
13677
13678 /* Macros for magic tty types. */
13679 #define isconsole(tp) ((tp) < tty_addr(NR_CONS))
13680 #define ispty(tp) ((tp) >= tty_addr(NR_CONS+NR_RS_LINES))
13681
13682 /* Macros for magic tty structure pointers. */
13683 #define FIRST_TTY tty_addr(0)
13684 #define END_TTY   tty_addr(sizeof(tty_table) / sizeof(tty_table[0]))
13685
13686 /* A device exists if at least its 'devread' function is defined. */
13687 #define tty_active(tp) ((tp)->tty_devread != NULL)
13688
13689 /* RS232 lines or pseudo terminals can be completely configured out. */

```

```

13690 #if NR_RS_LINES == 0
13691 #define rs_init(tp)      ((void) 0)
13692 #endif
13693 #if NR_PTYS == 0
13694 #define pty_init(tp)     ((void) 0)
13695 #define do_pty(tp, mp)   ((void) 0)
13696 #endif
13697
13698 FORWARD _PROTOTYPE( void tty_timed_out, (timer_t *tp) );;
13699 FORWARD _PROTOTYPE( void expire_timers, (void) );
13700 FORWARD _PROTOTYPE( void settimer, (tty_t *tty_ptr, int enable) );
13701 FORWARD _PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr) );
13702 FORWARD _PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr) );
13703 FORWARD _PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr) );
13704 FORWARD _PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr) );
13705 FORWARD _PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr) );
13706 FORWARD _PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr) );
13707 FORWARD _PROTOTYPE( void do_select, (tty_t *tp, message *m_ptr) );
13708 FORWARD _PROTOTYPE( void do_status, (message *m_ptr) );
13709 FORWARD _PROTOTYPE( void in_transfer, (tty_t *tp) );
13710 FORWARD _PROTOTYPE( int tty_echo, (tty_t *tp, int ch) );
13711 FORWARD _PROTOTYPE( void rawecho, (tty_t *tp, int ch) );
13712 FORWARD _PROTOTYPE( int back_over, (tty_t *tp) );
13713 FORWARD _PROTOTYPE( void reprint, (tty_t *tp) );
13714 FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp) );
13715 FORWARD _PROTOTYPE( void setattr, (tty_t *tp) );
13716 FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp) );
13717 FORWARD _PROTOTYPE( void tty_init, (void) );
13718
13719 /* Default attributes. */
13720 PRIVATE struct termios termios_defaults = {
13721     TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
13722     {
13723         TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
13724         TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
13725         TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
13726     },
13727 };
13728 PRIVATE struct winsize winsize_defaults; /* = all zeroes */
13729
13730 /* Global variables for the TTY task (declared extern in tty.h). */
13731 PUBLIC tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
13732 PUBLIC int ccurrent; /* currently active console */
13733 PUBLIC timer_t *tty_timers; /* queue of TTY timers */
13734 PUBLIC clock_t tty_next_timeout; /* time that the next alarm is due */
13735 PUBLIC struct machine machine; /* kernel environment variables */
13736
13737 /*=====
13738 *          tty_task
13739 *=====
13740 PUBLIC void main(void)
13741 {
13742     /* Main routine of the terminal task. */
13743
13744     message tty_mess; /* buffer for all incoming messages */
13745     unsigned line;
13746     int s;
13747     char *types[] = {"task", "driver", "server", "user"};
13748     register struct proc *rp;
13749     register tty_t *tp;
13750
13751     /* Initialize the TTY driver. */
13752     tty_init();
13753
13754     /* Get kernel environment (protected_mode, pc_at and ega are needed). */
13755     if (OK != (s=sys_getmachine(&machine))) {
13756         panic("TTY", "Couldn't obtain kernel environment.", s);
13757     }

```

```

13758
13759     /* Final one-time keyboard initialization. */
13760     kb_init_once();
13761
13762     printf("\n");
13763
13764     while (TRUE) {
13765
13766         /* Check for and handle any events on any of the ttys. */
13767         for (tp = FIRST_TTY; tp < END_TTY; tp++) {
13768             if (tp->tty_events) handle_events(tp);
13769         }
13770
13771         /* Get a request message. */
13772         receive(ANY, &tty_mess);
13773
13774         /* First handle all kernel notification types that the TTY supports.
13775          * - An alarm went off, expire all timers and handle the events.
13776          * - A hardware interrupt also is an invitation to check for events.
13777          * - A new kernel message is available for printing.
13778          * - Reset the console on system shutdown.
13779          * Then see if this message is different from a normal device driver
13780          * request and should be handled separately. These extra functions
13781          * do not operate on a device, in contrast to the driver requests.
13782          */
13783         switch (tty_mess.m_type) {
13784             case SYN_ALARM:           /* fall through */
13785                 expire_timers();    /* run watchdogs of expired timers */
13786                 continue;           /* continue to check for events */
13787             case HARD_INT: {        /* hardware interrupt notification */
13788                 if (tty_mess.NOTIFY_ARG & kbd_irq_set)
13789                     kbd_interrupt(&tty_mess); /* fetch chars from keyboard */
13790 #if NR_RS_LINES > 0
13791                 if (tty_mess.NOTIFY_ARG & rs_irq_set)
13792                     rs_interrupt(&tty_mess); /* serial I/O */
13793 #endif
13794                 expire_timers();    /* run watchdogs of expired timers */
13795                 continue;           /* continue to check for events */
13796             }
13797             case SYS_SIG: {         /* system signal */
13798                 sigset_t sigset = (sigset_t) tty_mess.NOTIFY_ARG;
13799
13800                 if (sigismember(&sigset, SIGHUP)) {
13801                     cons_stop();          /* switch to primary console */
13802                     if (irq_hook_id != -1) {
13803                         sys_irqdisable(&irq_hook_id);
13804                         sys_irqrmpolicy(KEYBOARD_IRQ, &irq_hook_id);
13805                     }
13806                 }
13807                 if (sigismember(&sigset, SIGTERM)) cons_stop();
13808                 if (sigismember(&sigset, SIGKILL)) do_new_kmess(&tty_mess);
13809                 continue;
13810             }
13811             case PANIC_DUMPS:       /* allow panic dumps */
13812                 cons_stop();          /* switch to primary console */
13813                 do_panic_dumps(&tty_mess);
13814                 continue;
13815             case DIAGNOSTICS:       /* a server wants to print some */
13816                 do_diagnostic(&tty_mess);
13817                 continue;
13818             case FKEY_CONTROL:      /* (un)register a fkey observer */
13819                 do_fkey_ctl(&tty_mess);
13820                 continue;
13821             default:                /* should be a driver request */
13822                 ;                   /* do nothing; end switch */
13823             }
13824
13825     /* Only device requests should get to this point. All requests,

```

```

13826     * except DEV_STATUS, have a minor device number. Check this
13827     * exception and get the minor device number otherwise.
13828     */
13829     if (tty_mess.m_type == DEV_STATUS) {
13830         do_status(&tty_mess);
13831         continue;
13832     }
13833     line = tty_mess.TTY_LINE;
13834     if ((line - CONS_MINOR) < NR_CONS) {
13835         tp = tty_addr(line - CONS_MINOR);
13836     } else if (line == LOG_MINOR) {
13837         tp = tty_addr(0);
13838     } else if ((line - RS232_MINOR) < NR_RS_LINES) {
13839         tp = tty_addr(line - RS232_MINOR + NR_CONS);
13840     } else if ((line - TTYPX_MINOR) < NR_PTYs) {
13841         tp = tty_addr(line - TTYPX_MINOR + NR_CONS + NR_RS_LINES);
13842     } else if ((line - PTYPX_MINOR) < NR_PTYs) {
13843         tp = tty_addr(line - PTYPX_MINOR + NR_CONS + NR_RS_LINES);
13844         if (tty_mess.m_type != DEV_IOCTL) {
13845             do_pty(tp, &tty_mess);
13846             continue;
13847         }
13848     } else {
13849         tp = NULL;
13850     }
13851
13852     /* If the device doesn't exist or is not configured return ENXIO. */
13853     if (tp == NULL || ! tty_active(tp)) {
13854         printf("Warning, TTY got illegal request %d from %d\n",
13855             tty_mess.m_type, tty_mess.m_source);
13856         tty_reply(TASK_REPLY, tty_mess.m_source,
13857                         tty_mess.PROC_NR, ENXIO);
13858         continue;
13859     }
13860
13861     /* Execute the requested device driver function. */
13862     switch (tty_mess.m_type) {
13863         case DEV_READ:      do_read(tp, &tty_mess);           break;
13864         case DEV_WRITE:     do_write(tp, &tty_mess);          break;
13865         case DEV_IOCTL:    do_ioctl(tp, &tty_mess);          break;
13866         case DEV_OPEN:      do_open(tp, &tty_mess);           break;
13867         case DEV_CLOSE:     do_close(tp, &tty_mess);          break;
13868         case DEV_SELECT:   do_select(tp, &tty_mess);          break;
13869         case CANCEL:       do_cancel(tp, &tty_mess);          break;
13870         default:
13871             printf("Warning, TTY got unexpected request %d from %d\n",
13872                 tty_mess.m_type, tty_mess.m_source);
13873             tty_reply(TASK_REPLY, tty_mess.m_source,
13874                         tty_mess.PROC_NR, EINVAL);
13875     }
13876 }
13877 }

13879 /*=====
13880 *                      do_status
13881 *=====
13882 PRIVATE void do_status(m_ptr)
13883 message *m_ptr;
13884 {
13885     register struct tty *tp;
13886     int event_found;
13887     int status;
13888     int ops;
13889
13890     /* Check for select or revive events on any of the ttys. If we found an,
13891     * event return a single status message for it. The FS will make another
13892     * call to see if there is more.

```

```

13893     */
13894     event_found = 0;
13895     for (tp = FIRST_TTY; tp < END_TTY; tp++) {
13896         if ((ops = select_try(tp, tp->tty_select_ops)) &&
13897             tp->tty_select_proc == m_ptr->m_source) {
13898
13899             /* I/O for a selected minor device is ready. */
13900             m_ptr->m_type = DEV_IO_READY;
13901             m_ptr->DEV_MINOR = tp->tty_index;
13902             m_ptr->DEV_SEL_OPS = ops;
13903
13904             tp->tty_select_ops &= ~ops;      /* unmark select event */
13905             event_found = 1;
13906             break;
13907         }
13908     else if (tp->tty_inrevived && tp->tty_incaller == m_ptr->m_source) {
13909
13910         /* Suspended request finished. Send a REVIVE. */
13911         m_ptr->m_type = DEV_REVIVE;
13912         m_ptr->REP_PROC_NR = tp->tty_inproc;
13913         m_ptr->REP_STATUS = tp->tty_incum;
13914
13915         tp->tty_inleft = tp->tty_incum = 0;
13916         tp->tty_inrevived = 0;           /* unmark revive event */
13917         event_found = 1;
13918         break;
13919     }
13920     else if (tp->tty_outrevived && tp->tty_outcaller == m_ptr->m_source) {
13921
13922         /* Suspended request finished. Send a REVIVE. */
13923         m_ptr->m_type = DEV_REVIVE;
13924         m_ptr->REP_PROC_NR = tp->tty_outproc;
13925         m_ptr->REP_STATUS = tp->tty_outcum;
13926
13927         tp->tty_outcum = 0;
13928         tp->tty_outrevived = 0;           /* unmark revive event */
13929         event_found = 1;
13930         break;
13931     }
13932 }
13933
13934 #if NR_PTYs > 0
13935     if (!event_found)
13936         event_found = pty_status(m_ptr);
13937 #endif
13938
13939     if (! event_found) {
13940         /* No events of interest were found. Return an empty message. */
13941         m_ptr->m_type = DEV_NO_STATUS;
13942     }
13943
13944     /* Almost done. Send back the reply message to the caller. */
13945     if ((status = send(m_ptr->m_source, m_ptr)) != OK) {
13946         panic("TTY", "send in do_status failed, status\n", status);
13947     }
13948 }

13950 /*=====
13951 *          do_read
13952 =====*/
13953 PRIVATE void do_read(tp, m_ptr)
13954 register tty_t *tp;          /* pointer to tty struct */
13955 register message *m_ptr;    /* pointer to message sent to the task */
13956 {
13957     /* A process wants to read from a terminal. */
13958     int r, status;
13959     phys_bytes phys_addr;

```

```

13960
13961     /* Check if there is already a process hanging in a read, check if the
13962     * parameters are correct, do I/O.
13963     */
13964     if (tp->tty_inleft > 0) {
13965         r = EIO;
13966     } else
13967     if (m_ptr->COUNT <= 0) {
13968         r = EINVAL;
13969     } else
13970     if (sys_umap(m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT,
13971             &phys_addr) != OK) {
13972         r = EFAULT;
13973     } else {
13974         /* Copy information from the message to the tty struct. */
13975         tp->tty_inrepcode = TASK_REPLY;
13976         tp->tty_incaller = m_ptr->m_source;
13977         tp->tty_inproc = m_ptr->PROC_NR;
13978         tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;
13979         tp->tty_inleft = m_ptr->COUNT;
13980
13981         if (!(tp->tty_termios.c_lflag & ICANON)
13982             && tp->tty_termios.c_cc[VTIME] > 0) {
13983             if (tp->tty_termios.c_cc[VMIN] == 0) {
13984                 /* MIN & TIME specify a read timer that finishes the
13985                 * read in TIME/10 seconds if no bytes are available.
13986                 */
13987                 settimer(tp, TRUE);
13988                 tp->tty_min = 1;
13989             } else {
13990                 /* MIN & TIME specify an inter-byte timer that may
13991                 * have to be cancelled if there are no bytes yet.
13992                 */
13993                 if (tp->tty_eotct == 0) {
13994                     settimer(tp, FALSE);
13995                     tp->tty_min = tp->tty_termios.c_cc[VMIN];
13996                 }
13997             }
13998         }
13999
14000         /* Anything waiting in the input buffer? Clear it out... */
14001         in_transfer(tp);
14002         /* ...then go back for more. */
14003         handle_events(tp);
14004         if (tp->tty_inleft == 0) {
14005             if (tp->tty_select_ops)
14006                 select_retry(tp);
14007             return;           /* already done */
14008         }
14009
14010         /* There were no bytes in the input queue available, so either suspend
14011         * the caller or break off the read if nonblocking.
14012         */
14013         if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
14014             r = EAGAIN;           /* cancel the read */
14015             tp->tty_inleft = tp->tty_incum = 0;
14016         } else {
14017             r = SUSPEND;          /* suspend the caller */
14018             tp->tty_inrepcode = REVIVE;
14019         }
14020     }
14021     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
14022     if (tp->tty_select_ops)
14023         select_retry(tp);
14024 }
```

```

14026 /*=====
14027 *          do_write
14028 *=====
14029 PRIVATE void do_write(tp, m_ptr)
14030 register tty_t *tp;
14031 register message *m_ptr;      /* pointer to message sent to the task */
14032 {
14033 /* A process wants to write on a terminal. */
14034     int r;
14035     phys_bytes phys_addr;
14036
14037 /* Check if there is already a process hanging in a write, check if the
14038 * parameters are correct, do I/O.
14039 */
14040     if (tp->tty_outleft > 0) {
14041         r = EIO;
14042     } else
14043     if (m_ptr->COUNT <= 0) {
14044         r = EINVAL;
14045     } else
14046     if (sys_umap(m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT,
14047           &phys_addr) != OK) {
14048         r = EFAULT;
14049     } else {
14050         /* Copy message parameters to the tty structure. */
14051         tp->tty_outrepcode = TASK_REPLY;
14052         tp->tty_outcaller = m_ptr->m_source;
14053         tp->tty_outproc = m_ptr->PROC_NR;
14054         tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
14055         tp->tty_outleft = m_ptr->COUNT;
14056
14057         /* Try to write. */
14058         handle_events(tp);
14059         if (tp->tty_outleft == 0)
14060             return; /* already done */
14061
14062         /* None or not all the bytes could be written, so either suspend the
14063         * caller or break off the write if nonblocking.
14064         */
14065         if (m_ptr->TTY_FLAGS & O_NONBLOCK) {           /* cancel the write */
14066             r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
14067             tp->tty_outleft = tp->tty_outcum = 0;
14068         } else {
14069             r = SUSPEND;                                /* suspend the caller */
14070             tp->tty_outrepcode = REVIVE;
14071         }
14072     }
14073     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
14074 }

14075 /*=====
14076 *          do_ioctl
14077 *=====
14078 PRIVATE void do_ioctl(tp, m_ptr)
14079 register tty_t *tp;
14080 register message *m_ptr;      /* pointer to message sent to task */
14081 {
14082 /* Perform an IOCTL on this terminal. Posix termios calls are handled
14083 * by the IOCTL system call
14084 */
14085
14086     int r;
14087     union {
14088         int i;
14089     } param;
14090     size_t size;
14091
14092

```

```

14093 /* Size of the ioctl parameter. */
14094 switch (m_ptr->TTY_REQUEST) {
14095     case TCGETS:           /* Posix tcgetattr function */
14096     case TCSETS:           /* Posix tcsetattr function, TCSANOW option */
14097     case TCSETSW:          /* Posix tcsetattr function, TCSADRAIN option */
14098     case TCSETSF:          /* Posix tcsetattr function, TCSAFLUSH option */
14099         size = sizeof(struct termios);
14100         break;
14101
14102     case TCSBRK:           /* Posix tcsendbreak function */
14103     case TCFLOW:           /* Posix tcflow function */
14104     case TCFLSH:           /* Posix tcflush function */
14105     case TIOCGPGRP:        /* Posix tcgetpgrp function */
14106     case TIOCSPGRP:        /* Posix tcsetpgrp function */
14107         size = sizeof(int);
14108         break;
14109
14110     case TIOCGWINSZ:       /* get window size (not Posix) */
14111     case TIOCSWINSZ:       /* set window size (not Posix) */
14112         size = sizeof(struct winsize);
14113         break;
14114
14115     case KIOCSMAP:          /* load keymap (Minix extension) */
14116         size = sizeof(keymap_t);
14117         break;
14118
14119     case TIOCSFON:          /* load font (Minix extension) */
14120         size = sizeof(u8_t [8192]);
14121         break;
14122
14123     case TCDRAIN:           /* Posix tcdrain function -- no parameter */
14124     default:                size = 0;
14125 }
14126
14127 r = OK;
14128 switch (m_ptr->TTY_REQUEST) {
14129     case TCGETS:
14130         /* Get the termios attributes. */
14131         r = sys_vircopy(SELF, D, (vir_bytes) &tp->tty_termios,
14132                         m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
14133                         (vir_bytes) size);
14134         break;
14135
14136     case TCSETSW:
14137     case TCSETSF:
14138     case TCDRAIN:
14139         if (tp->tty_outleft > 0) {
14140             /* Wait for all ongoing output processing to finish. */
14141             tp->tty_iocaller = m_ptr->m_source;
14142             tp->tty_ioproc = m_ptr->PROC_NR;
14143             tp->tty_ioreq = m_ptr->REQUEST;
14144             tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
14145             r = SUSPEND;
14146             break;
14147         }
14148         if (m_ptr->TTY_REQUEST == TCDRAIN) break;
14149         if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
14150         /*FALL THROUGH*/
14151     case TCSETS:
14152         /* Set the termios attributes. */
14153         r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
14154                         SELF, D, (vir_bytes) &tp->tty_termios, (vir_bytes) size);
14155         if (r != OK) break;
14156         setattr(tp);
14157         break;
14158
14159     case TCFLSH:

```

```

14160         r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
14161                 SELF, D, (vir_bytes) &param.i, (vir_bytes) size);
14162         if (r != OK) break;
14163         switch (param.i) {
14164             case TCIFLUSH:    tty_icancel(tp);                                break;
14165             case TCOFLUSH:   (*tp->tty_ocancel)(tp, 0);                      break;
14166             case TCIOFLUSH:  tty_icancel(tp); (*tp->tty_ocancel)(tp, 0); break;
14167             default:        r = EINVAL;
14168         }
14169     break;
14170
14171     case TCFLOW:
14172         r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
14173                 SELF, D, (vir_bytes) &param.i, (vir_bytes) size);
14174         if (r != OK) break;
14175         switch (param.i) {
14176             case TCOFF:
14177             case TCOON:
14178                 tp->tty_inhibited = (param.i == TCOFF);
14179                 tp->tty_events = 1;
14180                 break;
14181             case TCIOFF:
14182                 (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTOP]);
14183                 break;
14184             case TCION:
14185                 (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTART]);
14186                 break;
14187             default:
14188                 r = EINVAL;
14189         }
14190     break;
14191
14192     case TCSBRK:
14193         if (tp->tty_break != NULL) (*tp->tty_break)(tp,0);
14194         break;
14195
14196     case TIOCGWINSZ:
14197         r = sys_vircopy(SELF, D, (vir_bytes) &tp->tty_winsize,
14198                     m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
14199                     (vir_bytes) size);
14200     break;
14201
14202     case TIOCSWINSZ:
14203         r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
14204                 SELF, D, (vir_bytes) &tp->tty_winsize, (vir_bytes) size);
14205         /* SIGWINCH... */
14206         break;
14207
14208     case KIOCSMAP:
14209         /* Load a new keymap (only /dev/console). */
14210         if (isconsole(tp)) r = kbd_loadmap(m_ptr);
14211         break;
14212
14213     case TIOCSFON:
14214         /* Load a font into an EGA or VGA card (hs@hck.hr) */
14215         if (isconsole(tp)) r = con_loadfont(m_ptr);
14216         break;
14217
14218 /* These Posix functions are allowed to fail if _POSIX_JOB_CONTROL is
14219 * not defined.
14220 */
14221     case TIOCGPGRP:
14222     case TIOCSPGRP:
14223     default:
14224         r = ENOTTY;
14225     }
14226

```

```

14227 /* Send the reply. */
14228 tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
14229 }

14231 /*=====
14232 *          do_open
14233 *=====
14234 PRIVATE void do_open(tp, m_ptr)
14235 register tty_t *tp;
14236 message *m_ptr;           /* pointer to message sent to task */
14237 {
14238 /* A tty line has been opened. Make it the callers controlling tty if
14239 * O_NOCTTY is *not* set and it is not the log device. 1 is returned if
14240 * the tty is made the controlling tty, otherwise OK or an error code.
14241 */
14242     int r = OK;
14243
14244     if (m_ptr->TTY_LINE == LOG_MINOR) {
14245         /* The log device is a write-only diagnostics device. */
14246         if (m_ptr->COUNT & R_BIT) r = EACCES;
14247     } else {
14248         if (!(m_ptr->COUNT & O_NOCTTY)) {
14249             tp->tty_pgrp = m_ptr->PROC_NR;
14250             r = 1;
14251         }
14252         tp->tty_opencnt++;
14253     }
14254     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
14255 }

14257 /*=====
14258 *          do_close
14259 *=====
14260 PRIVATE void do_close(tp, m_ptr)
14261 register tty_t *tp;
14262 message *m_ptr;           /* pointer to message sent to task */
14263 {
14264 /* A tty line has been closed. Clean up the line if it is the last close. */
14265
14266     if (m_ptr->TTY_LINE != LOG_MINOR && --tp->tty_opencnt == 0) {
14267         tp->tty_pgrp = 0;
14268         tty_icancel(tp);
14269         (*tp->tty_ocancel)(tp, 0);
14270         (*tp->tty_close)(tp, 0);
14271         tp->tty_termios = termios_defaults;
14272         tp->tty_winsize = winsize_defaults;
14273         setattr(tp);
14274     }
14275     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, OK);
14276 }

14278 /*=====
14279 *          do_cancel
14280 *=====
14281 PRIVATE void do_cancel(tp, m_ptr)
14282 register tty_t *tp;
14283 message *m_ptr;           /* pointer to message sent to task */
14284 {
14285 /* A signal has been sent to a process that is hanging trying to read or write.
14286 * The pending read or write must be finished off immediately.
14287 */
14288     int proc_nr;
14289     int mode;
14290
14291     /* Check the parameters carefully, to avoid cancelling twice. */
14292     proc_nr = m_ptr->PROC_NR;

```

```

14294     mode = m_ptr->COUNT;
14295     if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
14296         /* Process was reading when killed. Clean up input. */
14297         tty_icancel(tp);
14298         tp->tty_inleft = tp->tty_incum = 0;
14299     }
14300     if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
14301         /* Process was writing when killed. Clean up output. */
14302         (*tp->tty_ocancel)(tp, 0);
14303         tp->tty_outleft = tp->tty_outcum = 0;
14304     }
14305     if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioproc) {
14306         /* Process was waiting for output to drain. */
14307         tp->tty_ioreq = 0;
14308     }
14309     tp->tty_events = 1;
14310     tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
14311 }

14313 PUBLIC int select_try(struct tty *tp, int ops)
14314 {
14315     int ready_ops = 0;
14316
14317     /* Special case. If line is hung up, no operations will block.
14318      * (and it can be seen as an exceptional condition.)
14319      */
14320     if (tp->tty_termios.c_ospeed == 80) {
14321         ready_ops |= ops;
14322     }
14323
14324     if (ops & SEL_RD) {
14325         /* Will i/o not block on read? */
14326         if (tp->tty_inleft > 0) {
14327             ready_ops |= SEL_RD; /* EIO - no blocking */
14328         } else if (tp->tty_incount > 0) {
14329             /* Is a regular read possible? tty_incount
14330              * says there is data. But a read will only succeed
14331              * in canonical mode if a newline has been seen.
14332              */
14333             if (!(tp->tty_termios.c_lflag & ICANON) ||
14334                 tp->tty_eotct > 0) {
14335                 ready_ops |= SEL_RD;
14336             }
14337         }
14338     }
14339
14340     if (ops & SEL_WR) {
14341         if (tp->tty_outleft > 0) ready_ops |= SEL_WR;
14342         else if ((*tp->tty_devwrite)(tp, 1)) ready_ops |= SEL_WR;
14343     }
14344
14345     return ready_ops;
14346 }

14348 PUBLIC int select_retry(struct tty *tp)
14349 {
14350     if (select_try(tp, tp->tty_select_ops))
14351         notify(tp->tty_select_proc);
14352     return OK;
14353 }

14355 /*=====
14356      *          handle_events
14357      *=====
14358 PUBLIC void handle_events(tp)
14359     tty_t *tp;           /* TTY to check for events. */
14360

```

```

14361 /* Handle any events pending on a TTY. These events are usually device
14362 * interrupts.
14363 *
14364 * Two kinds of events are prominent:
14365 * - a character has been received from the console or an RS232 line.
14366 * - an RS232 line has completed a write request (on behalf of a user).
14367 * The interrupt handler may delay the interrupt message at its discretion
14368 * to avoid swamping the TTY task. Messages may be overwritten when the
14369 * lines are fast or when there are races between different lines, input
14370 * and output, because MINIX only provides single buffering for interrupt
14371 * messages (in proc.c). This is handled by explicitly checking each line
14372 * for fresh input and completed output on each interrupt.
14373 */
14374 char *buf;
14375 unsigned count;
14376 int status;
14377
14378 do {
14379     tp->tty_events = 0;
14380
14381     /* Read input and perform input processing. */
14382     (*tp->tty_dehread)(tp, 0);
14383
14384     /* Perform output processing and write output. */
14385     (*tp->tty_devwrite)(tp, 0);
14386
14387     /* IOCTL waiting for some event? */
14388     if (tp->tty_ioreq != 0) dev_ioctl(tp);
14389 } while (tp->tty_events);
14390
14391 /* Transfer characters from the input queue to a waiting process. */
14392 in_transfer(tp);
14393
14394 /* Reply if enough bytes are available. */
14395 if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
14396     if (tp->tty_inrepcode == REVIVE) {
14397         notify(tp->tty_incaller);
14398         tp->tty_inrevived = 1;
14399     } else {
14400         tty_reply(tp->tty_inrepcode, tp->tty_incaller,
14401                 tp->tty_inproc, tp->tty_incum);
14402         tp->tty_inleft = tp->tty_incum = 0;
14403     }
14404 }
14405 if (tp->tty_select_ops)
14406     select_retry(tp);
14407 #if NR_PTYS > 0
14408     if (ispty(tp))
14409         select_retry_pty(tp);
14410 #endif
14411 }

14413 /*=====
14414 *          in_transfer
14415 *=====
14416 PRIVATE void in_transfer(tp)
14417 register tty_t *tp;           /* pointer to terminal to read from */
14418 {
14419 /* Transfer bytes from the input queue to a process reading from a terminal. */
14420
14421     int ch;
14422     int count;
14423     char buf[64], *bp;
14424
14425     /* Force read to succeed if the line is hung up, looks like EOF to reader. */
14426     if (tp->tty_termios.c_ospeed == 80) tp->tty_min = 0;
14427

```

```

14428 /* Anything to do? */
14429 if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;
14430
14431 bp = buf;
14432 while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
14433     ch = *tp->tty_intail;
14434
14435     if (!(ch & IN_EOF)) {
14436         /* One character to be delivered to the user. */
14437         *bp = ch & IN_CHAR;
14438         tp->tty_inleft--;
14439         if (++bp == bufend(buf)) {
14440             /* Temp buffer full, copy to user space. */
14441             sys_vircopy(SELF, D, (vir_bytes) buf,
14442                         tp->tty_inproc, D, tp->tty_in_vir,
14443                         (vir_bytes) buflen(buf));
14444             tp->tty_in_vir += buflen(buf);
14445             tp->tty_incum += buflen(buf);
14446             bp = buf;
14447         }
14448     }
14449
14450     /* Remove the character from the input queue. */
14451     if (++tp->tty_intail == bufend(tp->tty_inbuf))
14452         tp->tty_intail = tp->tty_inbuf;
14453     tp->tty_incount--;
14454     if (ch & IN_EOT) {
14455         tp->tty_eotct--;
14456         /* Don't read past a line break in canonical mode. */
14457         if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
14458     }
14459 }
14460
14461 if (bp > buf) {
14462     /* Leftover characters in the buffer. */
14463     count = bp - buf;
14464     sys_vircopy(SELF, D, (vir_bytes) buf,
14465                 tp->tty_inproc, D, tp->tty_in_vir, (vir_bytes) count);
14466     tp->tty_in_vir += count;
14467     tp->tty_incum += count;
14468 }
14469
14470 /* Usually reply to the reader, possibly even if incum == 0 (EOF). */
14471 if (tp->tty_inleft == 0) {
14472     if (tp->tty_inrepcode == REVIVE) {
14473         notify(tp->tty_incaller);
14474         tp->tty_inrevived = 1;
14475     } else {
14476         tty_reply(tp->tty_inrepcode, tp->tty_incaller,
14477                   tp->tty_inproc, tp->tty_incum);
14478         tp->tty_inleft = tp->tty_incum = 0;
14479     }
14480 }
14481 }

14482 /*=====
14483      *          in_process
14484      *=====
14485 PUBLIC int in_process(tp, buf, count)
14486 register tty_t *tp;           /* terminal on which character has arrived */
14487 char *buf;                  /* buffer with input characters */
14488 int count;                  /* number of input characters */
14489 {
14490 /* Characters have just been typed in. Process, save, and echo them. Return
14491 * the number of characters processed.
14492 */
14493 */
14494
14495     int ch, sig, ct;

```

```

14496 int timeset = FALSE;
14497 static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };
14498
14499 for (ct = 0; ct < count; ct++) {
14500     /* Take one character. */
14501     ch = *buf++ & BYTE;
14502
14503     /* Strip to seven bits? */
14504     if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;
14505
14506     /* Input extensions? */
14507     if (tp->tty_termios.c_lflag & IEXTEN) {
14508
14509         /* Previous character was a character escape? */
14510         if (tp->tty_escaped) {
14511             tp->tty_escaped = NOT_ESCAPED;
14512             ch |= IN_ESC; /* protect character */
14513         }
14514
14515         /* LNEXT (^V) to escape the next character? */
14516         if (ch == tp->tty_termios.c_cc[VLNEXT]) {
14517             tp->tty_escaped = ESCAPED;
14518             rawecho(tp, '^');
14519             rawecho(tp, '\b');
14520             continue; /* do not store the escape */
14521         }
14522
14523         /* REPRINT (^R) to reprint echoed characters? */
14524         if (ch == tp->tty_termios.c_cc[VREPRINT]) {
14525             reprint(tp);
14526             continue;
14527         }
14528     }
14529
14530     /* _POSIX_VDISABLE is a normal character value, so better escape it. */
14531     if (ch == _POSIX_VDISABLE) ch |= IN_ESC;
14532
14533     /* Map CR to LF, ignore CR, or map LF to CR. */
14534     if (ch == '\r') {
14535         if (tp->tty_termios.c_iflag & IGNCR) continue;
14536         if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
14537     } else
14538     if (ch == '\n') {
14539         if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
14540     }
14541
14542     /* Canonical mode? */
14543     if (tp->tty_termios.c_lflag & ICANON) {
14544
14545         /* Erase processing (rub out of last character). */
14546         if (ch == tp->tty_termios.c_cc[VERASE]) {
14547             (void) back_over(tp);
14548             if (!(tp->tty_termios.c_lflag & ECHOE)) {
14549                 (void) tty_echo(tp, ch);
14550             }
14551             continue;
14552         }
14553
14554         /* Kill processing (remove current line). */
14555         if (ch == tp->tty_termios.c_cc[VKILL]) {
14556             while (back_over(tp)) {
14557                 if (!(tp->tty_termios.c_lflag & ECHOE)) {
14558                     (void) tty_echo(tp, ch);
14559                     if (tp->tty_termios.c_lflag & ECHOK)
14560                         rawecho(tp, '\n');
14561                 }
14562             continue;

```

```

14563 }
14564
14565 /* EOF (`D) means end-of-file, an invisible "line break". */
14566 if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;
14567
14568 /* The line may be returned to the user after an LF. */
14569 if (ch == '\n') ch |= IN_EOT;
14570
14571 /* Same thing with EOL, whatever it may be. */
14572 if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
14573 }
14574
14575 /* Start/stop input control? */
14576 if (tp->tty_termios.c_iflag & IXON) {
14577
14578 /* Output stops on STOP (^S). */
14579 if (ch == tp->tty_termios.c_cc[VSTOP]) {
14580     tp->tty_inhibited = STOPPED;
14581     tp->tty_events = 1;
14582     continue;
14583 }
14584
14585 /* Output restarts on START (^Q) or any character if IXANY. */
14586 if (tp->tty_inhibited) {
14587     if (ch == tp->tty_termios.c_cc[VSTART]
14588         || (tp->tty_termios.c_iflag & IXANY)) {
14589         tp->tty_inhibited = RUNNING;
14590         tp->tty_events = 1;
14591         if (ch == tp->tty_termios.c_cc[VSTART])
14592             continue;
14593     }
14594 }
14595
14596 if (tp->tty_termios.c_lflag & ISIG) {
14597     /* Check for INTR (^?) and QUIT (^`) characters. */
14598     if (ch == tp->tty_termios.c_cc[VINTR]
14599         || ch == tp->tty_termios.c_cc[VQUIT]) {
14600         sig = SIGINT;
14601         if (ch == tp->tty_termios.c_cc[VQUIT]) sig = SIGQUIT;
14602         sigchar(tp, sig);
14603         (void) tty_echo(tp, ch);
14604         continue;
14605     }
14606 }
14607
14608 /* Is there space in the input buffer? */
14609 if (tp->tty_incount == buflen(tp->tty_inbuf)) {
14610     /* No space; discard in canonical mode, keep in raw mode. */
14611     if (tp->tty_termios.c_lflag & ICANON) continue;
14612     break;
14613 }
14614
14615 if (!(tp->tty_termios.c_lflag & ICANON)) {
14616     /* In raw mode all characters are "line breaks". */
14617     ch |= IN_EOT;
14618
14619     /* Start an inter-byte timer? */
14620     if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
14621         && tp->tty_termios.c_cc[VTIME] > 0) {
14622         settimer(tp, TRUE);
14623         timeset = TRUE;
14624     }
14625 }
14626
14627 /* Perform the intricate function of echoing. */
14628 if (tp->tty_termios.c_lflag & (ECHO|ECHONL)) ch = tty_echo(tp, ch);

```

```

14630
14631     /* Save the character in the input queue. */
14632     *tp->tty_inhead++ = ch;
14633     if (tp->tty_inhead == bufend(tp->tty_inbuf))
14634         tp->tty_inhead = tp->tty_inbuf;
14635     tp->tty_incount++;
14636     if (ch & IN_EOT) tp->tty_eotct++;
14637
14638     /* Try to finish input if the queue threatens to overflow. */
14639     if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
14640 }
14641 return ct;
14642 }

14644 /*=====
14645 *          echo
14646 *=====
14647 PRIVATE int tty_echo(tp, ch)
14648 register tty_t *tp;           /* terminal on which to echo */
14649 register int ch;             /* pointer to character to echo */
14650 {
14651 /* Echo the character if echoing is on. Some control characters are echoed
14652 * with their normal effect, other control characters are echoed as "'X",
14653 * normal characters are echoed normally. EOF (^D) is echoed, but immediately
14654 * backspaced over. Return the character with the echoed length added to its
14655 * attributes.
14656 */
14657     int len, rp;
14658
14659     ch &= ~IN_LEN;
14660     if (!(tp->tty_termios.c_lflag & ECHO)) {
14661         if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
14662                         & (ICANON|ECHONL)) == (ICANON|ECHONL))
14663             (*tp->tty_echo)(tp, '\n');
14664         return(ch);
14665     }
14666
14667 /* "Reprint" tells if the echo output has been messed up by other output. */
14668 rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;
14669
14670 if ((ch & IN_CHAR) < ' ')
14671     switch (ch & (IN_ESC|IN_EOF|IN_EOT|IN_CHAR)) {
14672         case '\t':
14673             len = 0;
14674             do {
14675                 (*tp->tty_echo)(tp, ' ');
14676                 len++;
14677             } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
14678             break;
14679         case '\r' | IN_EOT:
14680         case '\n' | IN_EOT:
14681             (*tp->tty_echo)(tp, ch & IN_CHAR);
14682             len = 0;
14683             break;
14684         default:
14685             (*tp->tty_echo)(tp, '^');
14686             (*tp->tty_echo)(tp, '@' + (ch & IN_CHAR));
14687             len = 2;
14688     }
14689 } else
14690 if ((ch & IN_CHAR) == '\177') {
14691     /* A DEL prints as "?". */
14692     (*tp->tty_echo)(tp, '^');
14693     (*tp->tty_echo)(tp, '?');
14694     len = 2;
14695 } else {
14696     (*tp->tty_echo)(tp, ch & IN_CHAR);
14697     len = 1;

```

```

14698     }
14699     if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo)(tp, '\b'); len--; }
14700
14701     tp->tty_reprint = rp;
14702     return(ch | (len << IN_LSHIFT));
14703 }

14705 /*=====
14706 *          rawecho
14707 *=====
14708 PRIVATE void rawecho(tp, ch)
14709 register tty_t *tp;
14710 int ch;
14711 {
14712 /* Echo without interpretation if ECHO is set. */
14713     int rp = tp->tty_reprint;
14714     if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo)(tp, ch);
14715     tp->tty_reprint = rp;
14716 }

14718 /*=====
14719 *          back_over
14720 *=====
14721 PRIVATE int back_over(tp)
14722 register tty_t *tp;
14723 {
14724 /* Backspace to previous character on screen and erase it. */
14725     u16_t *head;
14726     int len;
14727
14728     if (tp->tty_incount == 0) return(0); /* queue empty */
14729     head = tp->tty_inhead;
14730     if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
14731     if ((*--head & IN_EOT) return(0); /* can't erase "line breaks" */
14732     if (tp->tty_reprint) reprint(tp); /* reprint if messed up */
14733     tp->tty_inhead = head;
14734     tp->tty_incount--;
14735     if (tp->tty_termios.c_lflag & ECHOE) {
14736         len = (*head & IN_LEN) >> IN_LSHIFT;
14737         while (len > 0) {
14738             rawecho(tp, '\b');
14739             rawecho(tp, ' ');
14740             rawecho(tp, '\b');
14741             len--;
14742         }
14743     }
14744     return(1); /* one character erased */
14745 }

14747 /*=====
14748 *          reprint
14749 *=====
14750 PRIVATE void reprint(tp)
14751 register tty_t *tp; /* pointer to tty struct */
14752 {
14753 /* Restore what has been echoed to screen before if the user input has been
14754 * messed up by output, or if REPRINT ('R) is typed.
14755 */
14756     int count;
14757     u16_t *head;
14758
14759     tp->tty_reprint = FALSE;
14760
14761     /* Find the last line break in the input. */
14762     head = tp->tty_inhead;
14763     count = tp->tty_incount;
14764     while (count > 0) {

```

```

14765         if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
14766         if (head[-1] & IN_EOT) break;
14767         head--;
14768         count--;
14769     }
14770     if (count == tp->tty_incount) return;           /* no reason to reprint */
14771
14772     /* Show REPRINT ('R) and move to a new line. */
14773     (void) tty_echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
14774     rawecho(tp, '\r');
14775     rawecho(tp, '\n');
14776
14777     /* Reprint from the last break onwards. */
14778     do {
14779         if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
14780         *head = tty_echo(tp, *head);
14781         head++;
14782         count++;
14783     } while (count < tp->tty_incount);
14784 }

14785 /*=====
14786 *          out_process
14787 *=====
14788 */
14789 PUBLIC void out_process(tp, bstart, bpos, bend, ict, ocount)
14800 tty_t *tp;
14801 char *bstart, *bpos, *bend;      /* start/pos/end of circular buffer */
14802 int *ict;                      /* # input chars / input chars used */
14803 int *ocount;                   /* max output chars / output chars used */
14804 {
14805 /* Perform output processing on a circular buffer. *ict is the number of
14806 * bytes to process, and the number of bytes actually processed on return.
14807 * *ocount is the space available on input and the space used on output.
14808 * (Naturally *ict < *ocount.) The column position is updated modulo
14809 * the TAB size, because we really only need it for tabs.
14810 */
14811
14812     int tablen;
14813     int ict = *ict;
14814     int oct = *ocount;
14815     int pos = tp->tty_position;
14816
14817     while (ict > 0) {
14818         switch (*bpos) {
14819             case '\7':
14820                 break;
14821             case '\b':
14822                 pos--;
14823                 break;
14824             case '\r':
14825                 pos = 0;
14826                 break;
14827             case '\n':
14828                 if (((tp->tty_termios.c_oflag & (OPOST|ONLCR))
14829                               == (OPOST|ONLCR)) {
14830                     /* Map LF to CR+LF if there is space. Note that the
14831                     * next character in the buffer is overwritten, so
14832                     * we stop at this point.
14833                     */
14834                     if (oct >= 2) {
14835                         *bpos = '\r';
14836                         if (++bpos == bend) bpos = bstart;
14837                         *bpos = '\n';
14838                         pos = 0;
14839                         ict--;
14840                     }
14841                 }
14842             }
14843         }
14844     }
14845 }
```

```

14830          oct -= 2;
14831          }
14832          goto out_done; /* no space or buffer got changed */
14833      }
14834      break;
14835  case '\t':
14836      /* Best guess for the tab length. */
14837      tablen = TAB_SIZE - (pos & TAB_MASK);
14838
14839      if ((tp->tty_termios.c_oflag & (OPOST|XTABS))
14840          == (OPOST|XTABS)) {
14841          /* Tabs must be expanded. */
14842          if (oct >= tablen) {
14843              pos += tablen;
14844              ict--;
14845              oct -= tablen;
14846              do {
14847                  *bpos = ' ';
14848                  if (++bpos == bend) bpos = bstart;
14849              } while (--tablen != 0);
14850          }
14851          goto out_done;
14852      }
14853      /* Tabs are output directly. */
14854      pos += tablen;
14855      break;
14856  default:
14857      /* Assume any other character prints as one character. */
14858      pos++;
14859  }
14860  if (++bpos == bend) bpos = bstart;
14861  ict--;
14862  oct--;
14863 }
14864 out_done:
14865     tp->tty_position = pos & TAB_MASK;
14866
14867     *icount -= ict;      /* [io]ct are the number of chars not used */
14868     *ocount -= oct;      /* *[io]count are the number of chars that are used */
14869 }

14871 /*=====
14872  *          dev_ioctl1
14873  *=====
14874 PRIVATE void dev_ioctl(tp)
14875     tty_t *tp;
14876 {
14877 /* The ioctl's TCSETSW, TCSETSF and TCDRAIN wait for output to finish to make
14878 /* sure that an attribute change doesn't affect the processing of current
14879 /* output. Once output finishes the ioctl is executed as in do_ioctl().
14880 */
14881     int result;
14882
14883     if (tp->tty_outleft > 0) return;           /* output not finished */
14884
14885     if (tp->tty_ioreq != TCDRAIN) {
14886         if (tp->tty_ioreq == TCSETSF) tty_icancel(tp);
14887         result = sys_vircopy(tp->tty_ioproc, D, tp->tty_iovir,
14888                           SELF, D, (vir_bytes) &tp->tty_termios,
14889                           (vir_bytes) sizeof(tp->tty_termios));
14890         setattr(tp);
14891     }
14892     tp->tty_ioreq = 0;
14893     tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, result);
14894 }
```

```

14896 /*=====
14897 *          setattr
14898 *=====
14899 PRIVATE void setattr(tp)
14900 tty_t *tp;
14901 {
14902 /* Apply the new line attributes (raw/canonical, line speed, etc.) */
14903 u16_t *inp;
14904 int count;
14905
14906 if (!(tp->tty_termios.c_lflag & ICANON)) {
14907     /* Raw mode; put a "line break" on all characters in the input queue.
14908      * It is undefined what happens to the input queue when ICANON is
14909      * switched off, a process should use TCSAFLUSH to flush the queue.
14910      * Keeping the queue to preserve typeahead is the Right Thing, however
14911      * when a process does use TCSANOW to switch to raw mode.
14912     */
14913     count = tp->tty_eotct = tp->tty_incount;
14914     inp = tp->tty_intail;
14915     while (count > 0) {
14916         *inp |= IN_EOT;
14917         if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
14918         --count;
14919     }
14920 }
14921
14922 /* Inspect MIN and TIME. */
14923 settimer(tp, FALSE);
14924 if (tp->tty_termios.c_lflag & ICANON) {
14925     /* No MIN & TIME in canonical mode. */
14926     tp->tty_min = 1;
14927 } else {
14928     /* In raw mode MIN is the number of chars wanted, and TIME how long
14929      * to wait for them. With interesting exceptions if either is zero.
14930     */
14931     tp->tty_min = tp->tty_termios.c_cc[VMIN];
14932     if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
14933         tp->tty_min = 1;
14934 }
14935
14936 if (!(tp->tty_termios.c_iflag & IXON)) {
14937     /* No start/stop output control, so don't leave output inhibited. */
14938     tp->tty_inhibited = RUNNING;
14939     tp->tty_events = 1;
14940 }
14941
14942 /* Setting the output speed to zero hangs up the phone. */
14943 if (tp->tty_termios.c_ospeed == 80) sigchar(tp, SIGHUP);
14944
14945 /* Set new line speed, character size, etc at the device level. */
14946 (*tp->tty_ioctl)(tp, 0);
14947 }

14949 /*=====
14950 *          tty_reply
14951 *=====
14952 PUBLIC void tty_reply(code, replyee, proc_nr, status)
14953 int code;           /* TASK_REPLY or REVIVE */
14954 int replyee;        /* destination address for the reply */
14955 int proc_nr;        /* to whom should the reply go? */
14956 int status;         /* reply code */
14957 {
14958 /* Send a reply to a process that wanted to read or write data. */
14959     message tty_mess;
14960
14961     tty_mess.m_type = code;
14962     tty_mess.REP_PROC_NR = proc_nr;

```

```

14963     tty_mess.REP_STATUS = status;
14964
14965     if ((status = send(replyee, &tty_mess)) != OK) {
14966         panic("TTY","tty_reply failed, status\n", status);
14967     }
14968 }

14970 /*=====
14971 *          sigchar
14972 *=====
14973 PUBLIC void sigchar(tp, sig)
14974 register tty_t *tp;
14975 int sig;           /* SIGINT, SIGQUIT, SIGKILL or SIGHUP */
14976 {
14977 /* Process a SIGINT, SIGQUIT or SIGKILL char from the keyboard or SIGHUP from
14978 * a tty close, "stty 0", or a real RS-232 hangup. MM will send the signal to
14979 * the process group (INT, QUIT), all processes (KILL), or the session leader
14980 * (HUP).
14981 */
14982     int status;
14983
14984     if (tp->tty_pgrp != 0)
14985         if (OK != (status = sys_kill(tp->tty_pgrp, sig)))
14986             panic("TTY","Error, call to sys_kill failed", status);
14987
14988     if (!(tp->tty_termios.c_lflag & NOFLSH)) {
14989         tp->tty_incount = tp->tty_eotct = 0;    /* kill earlier input */
14990         tp->tty_intail = tp->tty_inhead;
14991         (*tp->tty_ocancel)(tp, 0);            /* kill all output */
14992         tp->tty_inhibited = RUNNING;
14993         tp->tty_events = 1;
14994     }
14995 }

14996 /*=====
14997 *          tty_icancel
14998 *=====
14999 PRIVATE void tty_icancel(tp)
15000 register tty_t *tp;
15001 {
15002 /* Discard all pending input, tty buffer or device. */
15003
15004     tp->tty_incount = tp->tty_eotct = 0;
15005     tp->tty_intail = tp->tty_inhead;
15006     (*tp->tty_icancel)(tp, 0);
15007 }

15008 */

15009 /*=====
15010 *          tty_init
15011 *=====
15012 PRIVATE void tty_init()
15013 {
15014 /* Initialize tty structure and call device initialization routines. */
15015
15016     register tty_t *tp;
15017     int s;
15018     struct sigaction sigact;
15019
15020     /* Initialize the terminal lines. */
15021     for (tp = FIRST_TTY,s=0; tp < END_TTY; tp++,s++) {
15022
15023         tp->tty_index = s;
15024
15025         tmr_inittimer(&tp->tty_tmr);
15026
15027         tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
15028         tp->tty_min = 1;
15029

```

```

15030     tp->tty_termios = termios_defaults;
15031     tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
15032                                         tty_devnop;
15033     if (tp < tty_addr(NR_CONS)) {
15034         scr_init(tp);
15035         tp->tty_minor = CONS_MINOR + s;
15036     } else
15037     if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
15038         rs_init(tp);
15039         tp->tty_minor = RS232_MINOR + s-NR_CONS;
15040     } else {
15041         pty_init(tp);
15042         tp->tty_minor = s - (NR_CONS+NR_RS_LINES) + TTYPX_MINOR;
15043     }
15044 }
15045 }

15047 /*=====
15048 *          tty_timed_out
15049 *=====
15050 PRIVATE void tty_timed_out(timer_t *tp)
15051 {
15052 /* This timer has expired. Set the events flag, to force processing. */
15053     tty_t *tty_ptr;
15054     tty_ptr = &tty_table[tmr_arg(tp)->ta_int];
15055     tty_ptr->tty_min = 0;           /* force read to succeed */
15056     tty_ptr->tty_events = 1;
15057 }

15059 /*=====
15060 *          expire_timers
15061 *=====
15062 PRIVATE void expire_timers(void)
15063 {
15064 /* A synchronous alarm message was received. Check if there are any expired
15065 * timers. Possibly set the event flag and reschedule another alarm.
15066 */
15067     clock_t now;                  /* current time */
15068     int s;

15070 /* Get the current time to compare the timers against. */
15071 if ((s=getuptime(&now)) != OK)
15072     panic("TTY","Couldn't get uptime from clock.", s);

15074 /* Scan the queue of timers for expired timers. This dispatch the watchdog
15075 * functions of expired timers. Possibly a new alarm call must be scheduled.
15076 */
15077 tmrs_exptimers(&tty_timers, now, NULL);
15078 if (tty_timers == NULL) tty_next_timeout = TMR_NEVER;
15079 else {                                /* set new sync alarm */
15080     tty_next_timeout = tty_timers->tmr_exp_time;
15081     if ((s=sys_setalarm(tty_next_timeout, 1)) != OK)
15082         panic("TTY","Couldn't set synchronous alarm.", s);
15083 }
15084 }

15086 /*=====
15087 *          settimer
15088 *=====
15089 PRIVATE void settimer(tty_ptr, enable)
15090 tty_t *tty_ptr;                      /* line to set or unset a timer on */
15091 int enable;                          /* set timer if true, otherwise unset */
15092 {
15093     clock_t now;                  /* current time */
15094     clock_t exp_time;
15095     int s;
15096

```

```

15097 /* Get the current time to calculate the timeout time. */
15098 if ((s=getuptime(&now)) != OK)
15099     panic("TTY","Couldn't get uptime from clock.", s);
15100 if (enable) {
15101     exp_time = now + tty_ptr->tty_termios.c_cc[VTIME] * (HZ/10);
15102     /* Set a new timer for enabling the TTY events flags. */
15103     tmrs_settimer(&tty_timers, &tty_ptr->tty_tmr,
15104                 exp_time, tty_timed_out, NULL);
15105 } else {
15106     /* Remove the timer from the active and expired lists. */
15107     tmrs_clrtimer(&tty_timers, &tty_ptr->tty_tmr, NULL);
15108 }
15109
15110 /* Now check if a new alarm must be scheduled. This happens when the front
15111 * of the timers queue was disabled or reinserted at another position, or
15112 * when a new timer was added to the front.
15113 */
15114 if (tty_timers == NULL) tty_next_timeout = TMR_NEVER;
15115 else if (tty_timers->tmr_exp_time != tty_next_timeout) {
15116     tty_next_timeout = tty_timers->tmr_exp_time;
15117     if ((s=sys_setalarm(tty_next_timeout, 1)) != OK)
15118         panic("TTY","Couldn't set synchronous alarm.", s);
15119 }
15120 }

15122 /*=====
15123 *          tty_devnop
15124 *=====
15125 PUBLIC int tty_devnop(tp, try)
15126 tty_t *tp;
15127 int try;
15128 {
15129     /* Some functions need not be implemented at the device level. */
15130 }

15132 /*=====
15133 *          do_select
15134 *=====
15135 PRIVATE void do_select(tp, m_ptr)
15136 register tty_t *tp;           /* pointer to tty struct */
15137 register message *m_ptr;      /* pointer to message sent to the task */
15138 {
15139     int ops, ready_ops = 0, watch;
15140
15141     ops = m_ptr->PROC_NR & (SEL_RD|SEL_WR|SEL_ERR);
15142     watch = (m_ptr->PROC_NR & SEL_NOTIFY) ? 1 : 0;
15143
15144     ready_ops = select_try(tp, ops);
15145
15146     if (!ready_ops && ops && watch) {
15147         tp->tty_select_ops |= ops;
15148         tp->tty_select_proc = m_ptr->m_source;
15149     }
15150
15151     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, ready_ops);
15152
15153     return;
15154 }

```

```
+++++
drivers/tty/keyboard.c
+++++
```

```
15200 /* Keyboard driver for PC's and AT's.
```

```

15201  /*
15202  * Changes:
15203  *   Jul 13, 2004  processes can observe function keys (Jorrit N. Herder)
15204  *   Jun 15, 2004  removed w reboot(), except panic dumps (Jorrit N. Herder)
15205  *   Feb 04, 1994  loadable keymaps (Marcus Hampel)
15206  */
15207
15208 #include "../drivers.h"
15209 #include <sys/time.h>
15210 #include <sys/select.h>
15211 #include <termios.h>
15212 #include <signal.h>
15213 #include <unistd.h>
15214 #include <minix/callnr.h>
15215 #include <minix/com.h>
15216 #include <minix/keymap.h>
15217 #include "tty.h"
15218 #include "keymaps/us-std.src"
15219 #include "../../kernel/const.h"
15220 #include "../../kernel/config.h"
15221 #include "../../kernel/type.h"
15222 #include "../../kernel/proc.h"
15223
15224 int irq_hook_id = -1;
15225
15226 /* Standard and AT keyboard. (PS/2 MCA implies AT throughout.) */
15227 #define KEYBD      0x60 /* I/O port for keyboard data */
15228
15229 /* AT keyboard. */
15230 #define KB_COMMAND    0x64 /* I/O port for commands on AT */
15231 #define KB_STATUS     0x64 /* I/O port for status on AT */
15232 #define KB_ACK        0xFA /* keyboard ack response */
15233 #define KB_OUT_FULL   0x01 /* status bit set when keypress char pending */
15234 #define KB_IN_FULL    0x02 /* status bit set when not ready to receive */
15235 #define LED_CODE       0xED /* command to keyboard to set LEDs */
15236 #define MAX_KB_ACK_RETRIES 0x1000 /* max #times to wait for kb ack */
15237 #define MAX_KB_BUSY_RETRIES 0x1000 /* max #times to loop while kb busy */
15238 #define KBIT          0x80 /* bit used to ack characters to keyboard */
15239
15240 /* Miscellaneous. */
15241 #define ESC_SCAN      0x01 /* reboot key when panicking */
15242 #define SLASH_SCAN    0x35 /* to recognize numeric slash */
15243 #define RSHIFT_SCAN   0x36 /* to distinguish left and right shift */
15244 #define HOME_SCAN     0x47 /* first key on the numeric keypad */
15245 #define INS_SCAN      0x52 /* INS for use in CTRL-ALT-INS reboot */
15246 #define DEL_SCAN      0x53 /* DEL for use in CTRL-ALT-DEL reboot */
15247
15248 #define CONSOLE        0 /* line number for console */
15249 #define KB_IN_BYTES    32 /* size of keyboard input buffer */
15250 PRIVATE char ibuf[KB_IN_BYTES]; /* input buffer */
15251 PRIVATE char *ihead = ibuf; /* next free spot in input buffer */
15252 PRIVATE char *itail = ibuf; /* scan code to return to TTY */
15253 PRIVATE int ictount; /* # codes in buffer */
15254
15255 PRIVATE int esc; /* escape scan code detected? */
15256 PRIVATE int alt_l; /* left alt key state */
15257 PRIVATE int alt_r; /* right alt key state */
15258 PRIVATE int alt; /* either alt key */
15259 PRIVATE int ctrl_l; /* left control key state */
15260 PRIVATE int ctrl_r; /* right control key state */
15261 PRIVATE int ctrl; /* either control key */
15262 PRIVATE int shift_l; /* left shift key state */
15263 PRIVATE int shift_r; /* right shift key state */
15264 PRIVATE int shift; /* either shift key */
15265 PRIVATE int num_down; /* num lock key depressed */
15266 PRIVATE int caps_down; /* caps lock key depressed */
15267 PRIVATE int scroll_down; /* scroll lock key depressed */
15268 PRIVATE int locks[NR_CONS]; /* per console lock keys state */

```

```

15269 /* Lock key active bits. Chosen to be equal to the keyboard LED bits. */
15270 #define SCROLL_LOCK      0x01
15271 #define NUM_LOCK        0x02
15272 #define CAPS_LOCK        0x04
15273
15274
15275 PRIVATE char numpad_map[] =
15276     {'H', 'Y', 'A', 'B', 'D', 'C', 'V', 'U', 'G', 'S', 'T', '@'};
15277
15278 /* Variables and definition for observed function keys. */
15279 typedef struct observer { int proc_nr; int events; } obs_t;
15280 PRIVATE obs_t fkey_obs[12]; /* observers for F1-F12 */
15281 PRIVATE obs_t sfkey_obs[12]; /* observers for SHIFT F1-F12 */
15282
15283 FORWARD _PROTOTYPE( int kb_ack, (void) );
15284 FORWARD _PROTOTYPE( int kb_wait, (void) );
15285 FORWARD _PROTOTYPE( int func_key, (int scode) );
15286 FORWARD _PROTOTYPE( int scan_keyboard, (void) );
15287 FORWARD _PROTOTYPE( unsigned make_break, (int scode) );
15288 FORWARD _PROTOTYPE( void set_leds, (void) );
15289 FORWARD _PROTOTYPE( void show_key_mappings, (void) );
15290 FORWARD _PROTOTYPE( int kb_read, (struct tty *tp, int try) );
15291 FORWARD _PROTOTYPE( unsigned map_key, (int scode) );
15292
15293 /*=====
15294 *          map_key0
15295 *=====
15296 /* Map a scan code to an ASCII code ignoring modifiers. */
15297 #define map_key0(scode) \
15298     ((unsigned) keymap[(scode) * MAP_COLS])
15299
15300 /*=====
15301 *          map_key
15302 *=====
15303 PRIVATE unsigned map_key(scode)
15304 int scode;
15305 {
15306 /* Map a scan code to an ASCII code. */
15307
15308     int caps, column, lk;
15309     u16_t *keyrow;
15310
15311     if (scode == SLASH_SCAN && esc) return '/'; /* don't map numeric slash */
15312
15313     keyrow = &keymap[scode * MAP_COLS];
15314
15315     caps = shift;
15316     lk = locks[ccurrent];
15317     if ((lk & NUM_LOCK) && HOME_SCAN <= scode && scode <= DEL_SCAN) caps = !caps;
15318     if ((lk & CAPS_LOCK) && (keyrow[0] & HASCAPS)) caps = !caps;
15319
15320     if (alt) {
15321         column = 2;
15322         if (ctrl || alt_r) column = 3; /* Ctrl + Alt == AltGr */
15323         if (caps) column = 4;
15324     } else {
15325         column = 0;
15326         if (caps) column = 1;
15327         if (ctrl) column = 5;
15328     }
15329     return keyrow[column] & ~HASCAPS;
15330 }
15331 /*=====
15332 *          kbd_interrupt
15333 *=====
15334 PUBLIC void kbd_interrupt(m_ptr)
15335 message *m_ptr;

```

```

15337 {
15338 /* A keyboard interrupt has occurred. Process it. */
15339 int scode;
15340 static timer_t timer;           /* timer must be static! */
15341
15342 /* Fetch the character from the keyboard hardware and acknowledge it. */
15343 scode = scan_keyboard();
15344
15345 /* Store the scancode in memory so the task can get at it later. */
15346 if (icount < KB_IN_BYTES) {
15347     *ihead++ = scode;
15348     if (ihead == ibuf + KB_IN_BYTES) ihead = ibuf;
15349     icount++;
15350     tty_table[ccurrent].tty_events = 1;
15351     if (tty_table[ccurrent].tty_select_ops & SEL_RD) {
15352         select_retry(&tty_table[ccurrent]);
15353     }
15354 }
15355 }

15357 /*=====
15358 *          kb_read
15359 *=====
15360 PRIVATE int kb_read(tp, try)
15361 tty_t *tp;
15362 int try;
15363 {
15364 /* Process characters from the circular keyboard buffer. */
15365     char buf[3];
15366     int scode;
15367     unsigned ch;
15368
15369     tp = &tty_table[ccurrent];           /* always use the current console */
15370
15371     if (try) {
15372         if (icount > 0) return 1;
15373         return 0;
15374     }
15375
15376     while (icount > 0) {
15377         scode = *itail++;                  /* take one key scan code */
15378         if (itail == ibuf + KB_IN_BYTES) itail = ibuf;
15379         icount--;
15380
15381         /* Function keys are being used for debug dumps. */
15382         if (func_key(scode)) continue;
15383
15384         /* Perform make/break processing. */
15385         ch = make_break(scode);
15386
15387         if (ch <= 0xFF) {
15388             /* A normal character. */
15389             buf[0] = ch;
15390             (void) in_process(tp, buf, 1);
15391         } else
15392             if (HOME <= ch && ch <= INSRT) {
15393                 /* An ASCII escape sequence generated by the numeric pad. */
15394                 buf[0] = ESC;
15395                 buf[1] = '[';
15396                 buf[2] = numpad_map[ch - HOME];
15397                 (void) in_process(tp, buf, 3);
15398             } else
15399                 if (ch == ALEFT) {
15400                     /* Choose lower numbered console as current console. */
15401                     select_console(ccurrent - 1);
15402                     set_leds();
15403                 } else
15404                     if (ch == ARIGHT) {

```

```

15405             /* Choose higher numbered console as current console. */
15406             select_console(ccurrent + 1);
15407             set_leds();
15408         } else
15409             if (AF1 <= ch && ch <= AF12) {
15410                 /* Alt-F1 is console, Alt-F2 is ttycl, etc. */
15411                 select_console(ch - AF1);
15412                 set_leds();
15413             } else
15414             if (CF1 <= ch && ch <= CF12) {
15415                 switch(ch) {
15416                     case CF1: show_key_mappings(); break;
15417                     case CF3: toggle_scroll(); break; /* hardware <-> software */
15418                     case CF7: sigchar(&tty_table[CONSOLE], SIGQUIT); break;
15419                     case CF8: sigchar(&tty_table[CONSOLE], SIGINT); break;
15420                     case CF9: sigchar(&tty_table[CONSOLE], SIGKILL); break;
15421                 }
15422             }
15423     }
15424
15425     return 1;
15426 }

15427 /*=====
15428 *                         make_break
15429 *=====
15430 */
15431 PRIVATE unsigned make_break(scode)
15432 int scode;           /* scan code of key just struck or released */
15433 {
15434 /* This routine can handle keyboards that interrupt only on key depression,
15435 * as well as keyboards that interrupt on key depression and key release.
15436 * For efficiency, the interrupt routine filters out most key releases.
15437 */
15438     int ch, make, escape;
15439     static int CAD_count = 0;
15440
15441 /* Check for CTRL-ALT-DEL, and if found, halt the computer. This would
15442 * be better done in keyboard() in case TTY is hung, except control and
15443 * alt are set in the high level code.
15444 */
15445 if (ctrl && alt && (scode == DEL_SCAN || scode == INS_SCAN))
15446 {
15447     if (++CAD_count == 3) sys_abort(RBT_HALTI);
15448     sys_kill(INIT_PROC_NR, SIGABRT);
15449     return -1;
15450 }
15451
15452 /* High-order bit set on key release. */
15453 make = (scode & KEY_RELEASE) == 0;           /* true if pressed */
15454
15455 ch = map_key(scode & ASCII_MASK);           /* map to ASCII */
15456
15457 escape = esc;      /* Key is escaped? (true if added since the XT) */
15458 esc = 0;
15459
15460 switch (ch) {
15461     case CTRL:          /* Left or right control key */
15462         *(escape ? &ctrl_r : &ctrl_l) = make;
15463         ctrl = ctrl_l | ctrl_r;
15464         break;
15465     case SHIFT:          /* Left or right shift key */
15466         *(scode == RSHIFT_SCAN ? &shift_r : &shift_l) = make;
15467         shift = shift_l | shift_r;
15468         break;
15469     case ALT:            /* Left or right alt key */
15470         *(escape ? &alt_r : &alt_l) = make;
15471         alt = alt_l | alt_r;
15472         break;

```

```

15473     case CALOCK:           /* Caps lock - toggle on 0 -> 1 transition */
15474         if (caps_down < make) {
15475             locks[ccurrent] ^= CAPS_LOCK;
15476             set_leds();
15477         }
15478         caps_down = make;
15479         break;
15480     case NLOCK:            /* Num lock */
15481         if (num_down < make) {
15482             locks[ccurrent] ^= NUM_LOCK;
15483             set_leds();
15484         }
15485         num_down = make;
15486         break;
15487     case SLOCK:             /* Scroll lock */
15488         if (scroll_down < make) {
15489             locks[ccurrent] ^= SCROLL_LOCK;
15490             set_leds();
15491         }
15492         scroll_down = make;
15493         break;
15494     case EXTKEY:            /* Escape keycode */
15495         esc = 1;                /* Next key is escaped */
15496         return(-1);
15497     default:                /* A normal key */
15498         if (make) return(ch);
15499     }
15500
15501 /* Key release, or a shift type key. */
15502 return(-1);
15503 }

15504 /*=====
15505 *                         set_leds
15506 *=====
15507 */
15508 PRIVATE void set_leds()
15509 {
15510 /* Set the LEDs on the caps, num, and scroll lock keys */
15511     int s;
15512     if (! machine.pc_at) return; /* PC/XT doesn't have LEDs */
15513
15514     kb_wait();               /* wait for buffer empty */
15515     if ((s=sys_outb(KEYBD, LED_CODE)) != OK)
15516         printf("Warning, sys_outb couldn't prepare for LED values: %d\n", s);
15517         /* prepare keyboard to accept LED values */
15518     kb_ack();                /* wait for ack response */
15519
15520     kb_wait();               /* wait for buffer empty */
15521     if ((s=sys_outb(KEYBD, locks[ccurrent])) != OK)
15522         printf("Warning, sys_outb couldn't give LED values: %d\n", s);
15523         /* give keyboard LED values */
15524     kb_ack();                /* wait for ack response */
15525 }

15526 /*=====
15527 *                         kb_wait
15528 *=====
15529 */
15530 PRIVATE int kb_wait()
15531 {
15532 /* Wait until the controller is ready; return zero if this times out. */
15533
15534     int retries, status, temp;
15535     int s;
15536
15537     retries = MAX_KB_BUSY_RETRIES + 1; /* wait until not busy */
15538     do {
15539         s = sys_inb(KB_STATUS, &status);

```

```

15540     if (status & KB_OUT_FULL) {
15541         s = sys_inb(KEYBD, &temp); /* discard value */
15542     }
15543     if (! (status & (KB_IN_FULL|KB_OUT_FULL)) )
15544         break; /* wait until ready */
15545     } while (--retries != 0); /* continue unless timeout */
15546     return(retries); /* zero on timeout, positive if ready */
15547 }

15549 /*=====
15550 *          kb_ack
15551 *=====
15552 PRIVATE int kb_ack()
15553 {
15554 /* Wait until kbd acknowledges last command; return zero if this times out. */
15555
15556     int retries, s;
15557     u8_t u8val;
15558
15559     retries = MAX_KB_ACK_RETRIES + 1;
15560     do {
15561         s = sys_inb(KEYBD, &u8val);
15562         if (u8val == KB_ACK)
15563             break; /* wait for ack */
15564     } while(--retries != 0); /* continue unless timeout */
15565
15566     return(retries); /* nonzero if ack received */
15567 }

15569 /*=====
15570 *          kb_init
15571 *=====
15572 PUBLIC void kb_init(tp)
15573     tty_t *tp;
15574 {
15575 /* Initialize the keyboard driver. */
15576
15577     tp->tty_devread = kb_read; /* input function */
15578 }

15580 /*=====
15581 *          kb_init_once
15582 *=====
15583 PUBLIC void kb_init_once(void)
15584 {
15585     int i;
15586
15587     set_leds(); /* turn off numlock led */
15588     scan_keyboard(); /* discard leftover keystroke */
15589
15590     /* Clear the function key observers array. Also see func_key(). */
15591     for (i=0; i<12; i++) {
15592         fkey_obs[i].proc_nr = NONE; /* F1-F12 observers */
15593         fkey_obs[i].events = 0; /* F1-F12 observers */
15594         sfkey_obs[i].proc_nr = NONE; /* Shift F1-F12 observers */
15595         sfkey_obs[i].events = 0; /* Shift F1-F12 observers */
15596     }
15597
15598     /* Set interrupt handler and enable keyboard IRQ. */
15599     irq_hook_id = KEYBOARD_IRQ; /* id to be returned on interrupt */
15600     if ((i=sys_irqsetpolicy(KEYBOARD_IRQ, IRQ_REENABLE, &irq_hook_id)) != OK)
15601         panic("TTY", "Couldn't set keyboard IRQ policy", i);
15602     if ((i=sys_irqenable(&irq_hook_id)) != OK)
15603         panic("TTY", "Couldn't enable keyboard IRQs", i);
15604     kbd_irq_set |= (1 << KEYBOARD_IRQ);
15605 }

```

```

15607 /*=====
15608 *          kbd_loadmap
15609 *=====
15610 PUBLIC int kbd_loadmap(m)
15611 message *m;
15612 {
15613 /* Load a new keymap. */
15614     int result;
15615     result = sys_vircopy(m->PROC_NR, D, (vir_bytes) m->ADDRESS,
15616         SELF, D, (vir_bytes) keymap,
15617         (vir_bytes) sizeof(keymap));
15618     return(result);
15619 }
15620 /*=====
15621 *          do_fkey_ctl
15622 *=====
15623 PUBLIC void do_fkey_ctl(m_ptr)
15624 message *m_ptr;           /* pointer to the request message */
15625 {
15626 /* This procedure allows processes to register a function key to receive
15627 * notifications if it is pressed. At most one binding per key can exist.
15628 */
15629     int i;
15630     int result;
15631
15632     switch (m_ptr->FKEY_REQUEST) {           /* see what we must do */
15633     case FKEY_MAP:                         /* request for new mapping */
15634         result = OK;                      /* assume everything will be ok*/
15635         for (i=0; i < 12; i++) {           /* check F1-F12 keys */
15636             if (bit_isset(m_ptr->FKEY_FKEYS, i+1) ) {
15637                 if (fkey_obs[i].proc_nr == NONE) {
15638                     fkey_obs[i].proc_nr = m_ptr->m_source;
15639                     fkey_obs[i].events = 0;
15640                     bit_unset(m_ptr->FKEY_FKEYS, i+1);
15641                 } else {
15642                     printf("WARNING, fkey_map failed F%d\n", i+1);
15643                     result = EBUSY;        /* report failure, but try rest */
15644                 }
15645             }
15646         }
15647         for (i=0; i < 12; i++) {           /* check Shift+F1-F12 keys */
15648             if (bit_isset(m_ptr->FKEY_SFKEYS, i+1) ) {
15649                 if (sfkey_obs[i].proc_nr == NONE) {
15650                     sfkey_obs[i].proc_nr = m_ptr->m_source;
15651                     sfkey_obs[i].events = 0;
15652                     bit_unset(m_ptr->FKEY_SFKEYS, i+1);
15653                 } else {
15654                     printf("WARNING, fkey_map failed Shift F%d\n", i+1);
15655                     result = EBUSY;        /* report failure but try rest */
15656                 }
15657             }
15658         }
15659     break;
15660     case FKEY_UNMAP:
15661         result = OK;                      /* assume everything will be ok*/
15662         for (i=0; i < 12; i++) {           /* check F1-F12 keys */
15663             if (bit_isset(m_ptr->FKEY_FKEYS, i+1) ) {
15664                 if (fkey_obs[i].proc_nr == m_ptr->m_source) {
15665                     fkey_obs[i].proc_nr = NONE;
15666                     fkey_obs[i].events = 0;
15667                     bit_unset(m_ptr->FKEY_FKEYS, i+1);
15668                 } else {
15669                     result = EPERM;        /* report failure, but try rest */
15670                 }
15671             }
15672         }
15673     }

```

```

15674     for (i=0; i < 12; i++) {           /* check Shift+F1-F12 keys */
15675         if (bit_isset(m_ptr->FKEY_SFKEYS, i+1) ) {
15676             if (sfkey_obs[i].proc_nr == m_ptr->m_source) {
15677                 sfkey_obs[i].proc_nr = NONE;
15678                 sfkey_obs[i].events = 0;
15679                 bit_unset(m_ptr->FKEY_SFKEYS, i+1);
15680             } else {
15681                 result = EPERM;      /* report failure, but try rest */
15682             }
15683         }
15684     }
15685     break;
15686 case FKEY_EVENTS:
15687     m_ptr->FKEY_FKEYS = m_ptr->FKEY_SFKEYS = 0;
15688     for (i=0; i < 12; i++) {           /* check (Shift+) F1-F12 keys */
15689         if (fkey_obs[i].proc_nr == m_ptr->m_source) {
15690             if (fkey_obs[i].events) {
15691                 bit_set(m_ptr->FKEY_FKEYS, i+1);
15692                 fkey_obs[i].events = 0;
15693             }
15694         }
15695         if (sfkey_obs[i].proc_nr == m_ptr->m_source) {
15696             if (sfkey_obs[i].events) {
15697                 bit_set(m_ptr->FKEY_SFKEYS, i+1);
15698                 sfkey_obs[i].events = 0;
15699             }
15700         }
15701     }
15702     break;
15703 default:
15704     result = EINVAL;                  /* key cannot be observed */
15705 }
15706
15707 /* Almost done, return result to caller. */
15708 m_ptr->m_type = result;
15709 send(m_ptr->m_source, m_ptr);
15710 }

15711 /*=====
15712 *          func_key
15713 *=====
15714 */
15715 PRIVATE int func_key(scode)
15716 int scode;                      /* scan code for a function key */
15717 {
15718 /* This procedure traps function keys for debugging purposes. Observers of
15719 * function keys are kept in a global array. If a subject (a key) is pressed
15720 * the observer is notified of the event. Initialization of the arrays is done
15721 * in kb_init, where NONE is set to indicate there is no interest in the key.
15722 * Returns FALSE on a key release or if the key is not observable.
15723 */
15724     message m;
15725     int key;
15726     int proc_nr;
15727     int i,s;
15728
15729     /* Ignore key releases. If this is a key press, get full key code. */
15730     if (scode & KEY_RELEASE) return(FALSE);        /* key release */
15731     key = map_key(scode);                         /* include modifiers */
15732
15733     /* Key pressed, now see if there is an observer for the pressed key.
15734     *      F1-F12    observers are in fkey_obs array.
15735     *      SHIFT   F1-F12  observers are in sfkey_req array.
15736     *      CTRL    F1-F12 reserved (see kb_read)
15737     *      ALT     F1-F12 reserved (see kb_read)
15738     * Other combinations are not in use. Note that Alt+Shift+F1-F12 is yet
15739     * defined in <minix/keymap.h>, and thus is easy for future extensions.
15740 */

```

```

15741     if (F1 <= key && key <= F12) {           /* F1-F12 */
15742         proc_nr = fkey_obs[key - F1].proc_nr;
15743         fkey_obs[key - F1].events ++ ;
15744     } else if (SF1 <= key && key <= SF12) {      /* Shift F2-F12 */
15745         proc_nr = sfkey_obs[key - SF1].proc_nr;
15746         sfkey_obs[key - SF1].events ++;
15747     }
15748     else {
15749         return(FALSE);                         /* not observable */
15750     }
15751
15752     /* See if an observer is registered and send it a message. */
15753     if (proc_nr != NONE) {
15754         m.NOTIFY_TYPE = FKEY_PRESSED;
15755         notify(proc_nr);
15756     }
15757     return(TRUE);
15758 }

15760 /*=====
15761 *                      show_key_mappings
15762 *=====
15763 PRIVATE void show_key_mappings()
15764 {
15765     int i,s;
15766     struct proc proc;
15767
15768     printf("\n");
15769     printf("System information. Known function key mappings to request debug dumps:\n");
15770     printf("-----\n");
15771     for (i=0; i<12; i++) {
15772
15773         printf(" %sF%d: ", i+1<10? " ":"", i+1);
15774         if (fkey_obs[i].proc_nr != NONE) {
15775             if ((s=sys_getproc(&proc, fkey_obs[i].proc_nr))!=OK)
15776                 printf("sys_getproc: %d\n", s);
15777             printf("%-14.14s", proc.p_name);
15778         } else {
15779             printf("%-14.14s", "<none>");
15780         }
15781
15782         printf(" %sShift-F%d: ", i+1<10? " ":"", i+1);
15783         if (sfkey_obs[i].proc_nr != NONE) {
15784             if ((s=sys_getproc(&proc, sfkey_obs[i].proc_nr))!=OK)
15785                 printf("sys_getproc: %d\n", s);
15786             printf("%-14.14s", proc.p_name);
15787         } else {
15788             printf("%-14.14s", "<none>");
15789         }
15790         printf("\n");
15791     }
15792     printf("\n");
15793     printf("Press one of the registered function keys to trigger a debug dump.\n");
15794     printf("\n");
15795 }

15796 /*=====
15797 *                      scan_keyboard
15798 *=====
15799 PRIVATE int scan_keyboard()
15800 {
15801     /* Fetch the character from the keyboard hardware and acknowledge it. */
15802     pvb_pair_t byte_in[2], byte_out[2];
15803
15804     byte_in[0].port = KEYBD;      /* get the scan code for the key struck */
15805     byte_in[1].port = PORT_B;    /* strobe the keyboard to ack the char */
15806     sys_vinb(byte_in, 2);       /* request actual input */

```

```

15808
15809     pv_set(byte_out[0], PORT_B, byte_in[1].value | KBIT); /* strobe bit high */
15810     pv_set(byte_out[1], PORT_B, byte_in[1].value);           /* then strobe low */
15811     sys_voutb(byte_out, 2);        /* request actual output */
15812
15813     return(byte_in[0].value);          /* return scan code */
15814 }

15816 /*=====
15817 *          do_panic_dumps
15818 *=====
15819 PUBLIC void do_panic_dumps(m)
15820 message *m;           /* request message to TTY */
15821 {
15822 /* Wait for keystrokes for printing debugging info and reboot. */
15823     int quiet, code;
15824
15825     /* A panic! Allow debug dumps until user wants to shutdown. */
15826     printf("\nHit ESC to reboot, DEL to shutdown, F-keys for debug dumps\n");
15827
15828     (void) scan_keyboard();      /* ack any old input */
15829     quiet = scan_keyboard();/* quiescent value (0 on PC, last code on AT)*/
15830     for (;;) {
15831         tickdelay(10);
15832         /* See if there are pending request for output, but don't block.
15833         * Diagnostics can span multiple printf()s, so do it in a loop.
15834         */
15835         while (nb_receive(ANY, m) == OK) {
15836             switch(m->m_type) {
15837                 case FKEY_CONTROL: do_fkey_ctl(m);      break;
15838                 case SYS_SIG:      do_new_kmess(m);    break;
15839                 case DIAGNOSTICS: do_diagnostic(m);   break;
15840                 default:          /* do nothing */ break;
15841             }
15842             tickdelay(1);           /* allow more */
15843         }
15844         code = scan_keyboard();
15845         if (code != quiet) {
15846             /* A key has been pressed. */
15847             switch (code) {           /* possibly abort MINIX */
15848                 case ESC_SCAN: sys_abort(RBT_REBOOT); return;
15849                 case DEL_SCAN:  sys_abort(RBT_HALTI); return;
15850             }
15851             (void) func_key(code); /* check for function key */
15852         }
15853     }
15854 }
15855 }

```

+++++  
drivers/tty/console.c  
+++++

```

15900 /* Code and data for the IBM console driver.
15901 *
15902 * The 6845 video controller used by the IBM PC shares its video memory with
15903 * the CPU somewhere in the 0xB0000 memory bank. To the 6845 this memory
15904 * consists of 16-bit words. Each word has a character code in the low byte
15905 * and a so-called attribute byte in the high byte. The CPU directly modifies
15906 * video memory to display characters, and sets two registers on the 6845 that
15907 * specify the video origin and the cursor position. The video origin is the
15908 * place in video memory where the first character (upper left corner) can
15909 * be found. Moving the origin is a fast way to scroll the screen. Some
15910 * video adapters wrap around the top of video memory, so the origin can

```

```

15911 * move without bounds. For other adapters screen memory must sometimes be
15912 * moved to reset the origin. All computations on video memory use character
15913 * (word) addresses for simplicity and assume there is no wrapping. The
15914 * assembly support functions translate the word addresses to byte addresses
15915 * and the scrolling function worries about wrapping.
15916 */
15917
15918 #include "../drivers.h"
15919 #include <termios.h>
15920 #include <minix/callnr.h>
15921 #include <minix/com.h>
15922 #include "tty.h"
15923
15924 #include "../../kernel/const.h"
15925 #include "../../kernel/config.h"
15926 #include "../../kernel/type.h"
15927
15928 /* Definitions used by the console driver. */
15929 #define MONO_BASE 0xB0000L /* base of mono video memory */
15930 #define COLOR_BASE 0xB8000L /* base of color video memory */
15931 #define MONO_SIZE 0x1000 /* 4K mono video memory */
15932 #define COLOR_SIZE 0x4000 /* 16K color video memory */
15933 #define EGA_SIZE 0x8000 /* EGA & VGA have at least 32K */
15934 #define BLANK_COLOR 0x0700 /* determines cursor color on blank screen */
15935 #define SCROLL_UP 0 /* scroll forward */
15936 #define SCROLL_DOWN 1 /* scroll backward */
15937 #define BLANK_MEM ((u16_t *) 0) /* tells mem_vid_copy() to blank the screen */
15938 #define CONS_RAM_WORDS 80 /* video ram buffer size */
15939 #define MAX_ESC_PARMS 4 /* number of escape sequence params allowed */
15940
15941 /* Constants relating to the controller chips. */
15942 #define M_6845 0x3B4 /* port for 6845 mono */
15943 #define C_6845 0x3D4 /* port for 6845 color */
15944 #define INDEX 0 /* 6845's index register */
15945 #define DATA 1 /* 6845's data register */
15946 #define STATUS 6 /* 6845's status register */
15947 #define VID_ORG 12 /* 6845's origin register */
15948 #define CURSOR 14 /* 6845's cursor register */
15949
15950 /* Beeper. */
15951 #define BEEP_FREQ 0x0533 /* value to put into timer to set beep freq */
15952 #define B_TIME 3 /* length of CTRL-G beep is ticks */
15953
15954 /* definitions used for font management */
15955 #define GA_SEQUENCER_INDEX 0x3C4
15956 #define GA_SEQUENCER_DATA 0x3C5
15957 #define GA_GRAPHICS_INDEX 0x3CE
15958 #define GA_GRAPHICS_DATA 0x3CF
15959 #define GA_VIDEO_ADDRESS 0xA0000L
15960 #define GA_FONT_SIZE 8192
15961
15962 /* Global variables used by the console driver and assembly support. */
15963 PUBLIC int vid_index; /* index of video segment in remote mem map */
15964 PUBLIC u16_t vid_seg;
15965 PUBLIC vir_bytes vid_off; /* video ram is found at vid_seg:vid_off */
15966 PUBLIC unsigned vid_size; /* 0x2000 for color or 0x0800 for mono */
15967 PUBLIC unsigned vid_mask; /* 0xFFFF for color or 0x07FF for mono */
15968 PUBLIC unsigned blank_color = BLANK_COLOR; /* display code for blank */
15969
15970 /* Private variables used by the console driver. */
15971 PRIVATE int vid_port; /* I/O port for accessing 6845 */
15972 PRIVATE int wrap; /* hardware can wrap? */
15973 PRIVATE int softscroll; /* 1 = software scrolling, 0 = hardware */
15974 PRIVATE int beeping; /* speaker is beeping? */
15975 PRIVATE unsigned font_lines; /* font lines per character */
15976 PRIVATE unsigned scr_width; /* # characters on a line */
15977 PRIVATE unsigned scr_lines; /* # lines on the screen */

```

```

15978 PRIVATE unsigned scr_size;           /* # characters on the screen */
15979
15980 /* Per console data. */
15981 typedef struct console {
15982     tty_t *c_tty;                      /* associated TTY struct */
15983     int c_column;                     /* current column number (0-origin) */
15984     int c_row;                       /* current row (0 at top of screen) */
15985     int c_rwords;                    /* number of WORDS (not bytes) in outqueue */
15986     unsigned c_start;                /* start of video memory of this console */
15987     unsigned c_limit;                /* limit of this console's video memory */
15988     unsigned c_org;                  /* location in RAM where 6845 base points */
15989     unsigned c_cur;                  /* current position of cursor in video RAM */
15990     unsigned c_attr;                /* character attribute */
15991     unsigned c_blank;                /* blank attribute */
15992     char c_reverse;                 /* reverse video */
15993     char c_esc_state;              /* 0=normal, 1=ESC, 2=ESC[ */
15994     char c_esc_intro;              /* Distinguishing character following ESC */
15995     int *c_esc_parmp;              /* pointer to current escape parameter */
15996     int c_esc_parmv[MAX_ESC_PARMS]; /* list of escape parameters */
15997     u16_t c_ramqueue[CONS_RAM_WORDS]; /* buffer for video RAM */
15998 } console_t;
15999
16000 PRIVATE int nr_cons= 1;             /* actual number of consoles */
16001 PRIVATE console_t cons_table[NR_CONS];
16002 PRIVATE console_t *curcons;        /* currently visible */
16003
16004 /* Color if using a color controller. */
16005 #define color (vid_port == C_6845)
16006
16007 /* Map from ANSI colors to the attributes used by the PC */
16008 PRIVATE int ansi_colors[8] = {0, 4, 2, 6, 1, 5, 3, 7};
16009
16010 /* Structure used for font management */
16011 struct sequence {
16012     unsigned short index;
16013     unsigned char port;
16014     unsigned char value;
16015 };
16016
16017 FORWARD _PROTOTYPE( int cons_write, (struct tty *tp, int try) );
16018 FORWARD _PROTOTYPE( void cons_echo, (tty_t *tp, int c) );
16019 FORWARD _PROTOTYPE( void out_char, (console_t *cons, int c) );
16020 FORWARD _PROTOTYPE( void putk, (int c) );
16021 FORWARD _PROTOTYPE( void beep, (void) );
16022 FORWARD _PROTOTYPE( void do_escape, (console_t *cons, int c) );
16023 FORWARD _PROTOTYPE( void flush, (console_t *cons) );
16024 FORWARD _PROTOTYPE( void parse_escape, (console_t *cons, int c) );
16025 FORWARD _PROTOTYPE( void scroll_screen, (console_t *cons, int dir) );
16026 FORWARD _PROTOTYPE( void set_6845, (int reg, unsigned val) );
16027 FORWARD _PROTOTYPE( void get_6845, (int reg, unsigned *val) );
16028 FORWARD _PROTOTYPE( void stop_beep, (timer_t *tmrp) );
16029 FORWARD _PROTOTYPE( void cons_org0, (void) );
16030 FORWARD _PROTOTYPE( int ga_program, (struct sequence *seq) );
16031 FORWARD _PROTOTYPE( int cons_ioctl, (tty_t *tp, int) );
16032
16033 /*=====
16034 *                               cons_write
16035 *=====
16036 PRIVATE int cons_write(tp, try)
16037 register struct tty *tp;          /* tells which terminal is to be used */
16038 int try;
16039 {
16040 /* Copy as much data as possible to the output queue, then start I/O. On
16041 * memory-mapped terminals, such as the IBM console, the I/O will also be
16042 * finished, and the counts updated. Keep repeating until all I/O done.
16043 */
16044

```

```

16045     int count;
16046     int result;
16047     register char *tbuf;
16048     char buf[64];
16049     console_t *cons = tp->tty_priv;
16050
16051     if (try) return 1; /* we can always write to console */
16052
16053     /* Check quickly for nothing to do, so this can be called often without
16054      * unmodular tests elsewhere.
16055      */
16056     if ((count = tp->tty_outleft) == 0 || tp->tty_inhibited) return;
16057
16058     /* Copy the user bytes to buf[] for decent addressing. Loop over the
16059      * copies, since the user buffer may be much larger than buf[].
16060      */
16061     do {
16062         if (count > sizeof(buf)) count = sizeof(buf);
16063         if ((result = sys_vircopy(tp->tty_outproc, D, tp->tty_out_vir,
16064                               SELF, D, (vir_bytes) buf, (vir_bytes) count)) != OK)
16065             break;
16066         tbuf = buf;
16067
16068         /* Update terminal data structure. */
16069         tp->tty_out_vir += count;
16070         tp->tty_outcum += count;
16071         tp->tty_outleft -= count;
16072
16073         /* Output each byte of the copy to the screen. Avoid calling
16074          * out_char() for the "easy" characters, put them into the buffer
16075          * directly.
16076          */
16077         do {
16078             if ((unsigned) *tbuf < ' ' || cons->c_esc_state > 0
16079                         || cons->c_column >= scr_width
16080                         || cons->c_rwords >= buflen(cons->c_ramqueue))
16081             {
16082                 out_char(cons, *tbuf++);
16083             } else {
16084                 cons->c_ramqueue[cons->c_rwords++] =
16085                             cons->c_attr | (*tbuf++ & BYTE);
16086                 cons->c_column++;
16087             }
16088         } while (--count != 0);
16089     } while ((count = tp->tty_outleft) != 0 && !tp->tty_inhibited);
16090
16091     flush(cons);           /* transfer anything buffered to the screen */
16092
16093     /* Reply to the writer if all output is finished or if an error occurred. */
16094     if (tp->tty_outleft == 0 || result != OK) {
16095         /* REVIVE is not possible. I/O on memory mapped consoles finishes. */
16096         tty_reply(tp->tty_outproc, tp->tty_outcaller, tp->tty_outproc,
16097                   tp->tty_outcum);
16098         tp->tty_outcum = 0;
16099     }
16100 }

16102 /*=====
16103 *                               cons_echo
16104 *=====
16105 PRIVATE void cons_echo(tp, c)
16106 register tty_t *tp;           /* pointer to tty struct */
16107 int c;                      /* character to be echoed */
16108 {
16109     /* Echo keyboard input (print & flush). */
16110     console_t *cons = tp->tty_priv;
16111

```

```

16112     out_char(cons, c);
16113     flush(cons);
16114 }

16115 /*=====
16116 *          out_char
16117 *=====
16118 */
16119 PRIVATE void out_char(cons, c)
16120 register console_t *cons;      /* pointer to console struct */
16121 int c;                      /* character to be output */
16122 {
16123 /* Output a character on the console. Check for escape sequences first. */
16124 if (cons->c_esc_state > 0) {
16125     parse_escape(cons, c);
16126     return;
16127 }
16128
16129 switch(c) {
16130     case 000:           /* null is typically used for padding */
16131         return;          /* better not do anything */
16132
16133     case 007:           /* ring the bell */
16134         flush(cons);    /* print any chars queued for output */
16135         beep();
16136         return;
16137
16138     case '\b':           /* backspace */
16139         if (--cons->c_column < 0) {
16140             if (--cons->c_row >= 0) cons->c_column += scr_width;
16141         }
16142         flush(cons);
16143         return;
16144
16145     case '\n':           /* line feed */
16146         if (((cons->c_tty->tty_termios.c_oflag & (OPOST|ONLCR))
16147               == (OPOST|ONLCR)) {
16148             cons->c_column = 0;
16149         }
16150         /*FALL THROUGH*/
16151     case 013:             /* CTRL-K */
16152     case 014:             /* CTRL-L */
16153         if (cons->c_row == scr_lines-1) {
16154             scroll_screen(cons, SCROLL_UP);
16155         } else {
16156             cons->c_row++;
16157         }
16158         flush(cons);
16159         return;
16160
16161     case '\r':           /* carriage return */
16162         cons->c_column = 0;
16163         flush(cons);
16164         return;
16165
16166     case '\t':           /* tab */
16167         cons->c_column = (cons->c_column + TAB_SIZE) & ~TAB_MASK;
16168         if (cons->c_column > scr_width) {
16169             cons->c_column -= scr_width;
16170             if (cons->c_row == scr_lines-1) {
16171                 scroll_screen(cons, SCROLL_UP);
16172             } else {
16173                 cons->c_row++;
16174             }
16175         }
16176         flush(cons);
16177         return;
16178

```

```

16179     case 033:          /* ESC - start of an escape sequence */
16180         flush(cons);    /* print any chars queued for output */
16181         cons->c_esc_state = 1; /* mark ESC as seen */
16182         return;
16183
16184     default:           /* printable chars are stored in ramqueue */
16185         if (cons->c_column >= scr_width) {
16186             if (!LINEWRAP) return;
16187             if (cons->c_row == scr_lines-1) {
16188                 scroll_screen(cons, SCROLL_UP);
16189             } else {
16190                 cons->c_row++;
16191             }
16192             cons->c_column = 0;
16193             flush(cons);
16194         }
16195         if (cons->c_rwords == buflen(cons->c_ramqueue)) flush(cons);
16196         cons->c_ramqueue[cons->c_rwords++] = cons->c_attr | (c & BYTE);
16197         cons->c_column++;           /* next column */
16198         return;
16199     }
16200 }
16201 /*=====
16202 *          scroll_screen
16203 *=====
16204 PRIVATE void scroll_screen(cons, dir)
16205 register console_t *cons;      /* pointer to console struct */
16206 int dir;                      /* SCROLL_UP or SCROLL_DOWN */
16207 {
16208     unsigned new_line, new_org, chars;
16209
16210     flush(cons);
16211     chars = scr_size - scr_width;        /* one screen minus one line */
16212
16213     /* Scrolling the screen is a real nuisance due to the various incompatible
16214     * video cards. This driver supports software scrolling (Hercules?),
16215     * hardware scrolling (mono and CGA cards) and hardware scrolling without
16216     * wrapping (EGA cards). In the latter case we must make sure that
16217     * c_start <= c_org && c_org + scr_size <= c_limit
16218     * holds, because EGA doesn't wrap around the end of video memory.
16219     */
16220     if (dir == SCROLL_UP) {
16221         /* Scroll one line up in 3 ways: soft, avoid wrap, use origin. */
16222         if (softscroll) {
16223             vid_vid_copy(cons->c_start + scr_width, cons->c_start, chars);
16224         } else
16225             if (!wrap && cons->c_org + scr_size + scr_width >= cons->c_limit) {
16226                 vid_vid_copy(cons->c_org + scr_width, cons->c_start, chars);
16227                 cons->c_org = cons->c_start;
16228             } else {
16229                 cons->c_org = (cons->c_org + scr_width) & vid_mask;
16230             }
16231             new_line = (cons->c_org + chars) & vid_mask;
16232     } else {
16233         /* Scroll one line down in 3 ways: soft, avoid wrap, use origin. */
16234         if (softscroll) {
16235             vid_vid_copy(cons->c_start, cons->c_start + scr_width, chars);
16236         } else
16237             if (!wrap && cons->c_org < cons->c_start + scr_width) {
16238                 new_org = cons->c_limit - scr_size;
16239                 vid_vid_copy(cons->c_org, new_org + scr_width, chars);
16240                 cons->c_org = new_org;
16241             } else {
16242                 cons->c_org = (cons->c_org - scr_width) & vid_mask;
16243             }
16244             new_line = cons->c_org;
16245     }

```

```

16246     }
16247     /* Blank the new line at top or bottom. */
16248     blank_color = cons->c_blank;
16249     mem_vid_copy(BLANK_MEM, new_line, scr_width);
16250
16251     /* Set the new video origin. */
16252     if (cons == curcons) set_6845(VID_ORG, cons->c_org);
16253     flush(cons);
16254 }

16256 /*=====
16257 *          flush
16258 *=====
16259 PRIVATE void flush(cons)
16260 register console_t *cons;      /* pointer to console struct */
16261 {
16262 /* Send characters buffered in 'ramqueue' to screen memory, check the new
16263 * cursor position, compute the new hardware cursor position and set it.
16264 */
16265     unsigned cur;
16266     tty_t *tp = cons->c_tty;
16267
16268 /* Have the characters in 'ramqueue' transferred to the screen. */
16269     if (cons->c_rwords > 0) {
16270         mem_vid_copy(cons->c_ramqueue, cons->c_cur, cons->c_rwords);
16271         cons->c_rwords = 0;
16272
16273         /* TTY likes to know the current column and if echoing messed up. */
16274         tp->tty_position = cons->c_column;
16275         tp->tty_reprint = TRUE;
16276     }
16277
16278 /* Check and update the cursor position. */
16279     if (cons->c_column < 0) cons->c_column = 0;
16280     if (cons->c_column > scr_width) cons->c_column = scr_width;
16281     if (cons->c_row < 0) cons->c_row = 0;
16282     if (cons->c_row >= scr_lines) cons->c_row = scr_lines - 1;
16283     cur = cons->c_org + cons->c_row * scr_width + cons->c_column;
16284     if (cur != cons->c_cur) {
16285         if (cons == curcons) set_6845(CURSOR, cur);
16286         cons->c_cur = cur;
16287     }
16288 }

16290 /*=====
16291 *          parse_escape
16292 *=====
16293 PRIVATE void parse_escape(cons, c)
16294 register console_t *cons;      /* pointer to console struct */
16295 char c;                      /* next character in escape sequence */
16296 {
16297 /* The following ANSI escape sequences are currently supported.
16298 * If n and/or m are omitted, they default to 1.
16299 *   ESC [nA moves up n lines
16300 *   ESC [nB moves down n lines
16301 *   ESC [nC moves right n spaces
16302 *   ESC [nD moves left n spaces
16303 *   ESC [m;nH" moves cursor to (m,n)
16304 *   ESC [J clears screen from cursor
16305 *   ESC [K clears line from cursor
16306 *   ESC [nL inserts n lines at cursor
16307 *   ESC [nM deletes n lines at cursor
16308 *   ESC [nP deletes n chars at cursor
16309 *   ESC [n@ inserts n chars at cursor
16310 *   ESC [nm enables rendition n (0=normal, 4=bold, 5=blinking, 7=reverse)
16311 *   ESC M scrolls the screen backwards if the cursor is on the top line
16312 */

```

```

16313
16314     switch (cons->c_esc_state) {
16315         case 1: /* ESC seen */
16316             cons->c_esc_intro = '\0';
16317             cons->c_esc_parmp = bufend(cons->c_esc_parmv);
16318             do {
16319                 --cons->c_esc_parmp = 0;
16320             } while (cons->c_esc_parmp > cons->c_esc_parmv);
16321             switch (c) {
16322                 case '[': /* Control Sequence Introducer */
16323                     cons->c_esc_intro = c;
16324                     cons->c_esc_state = 2;
16325                     break;
16326                 case 'M': /* Reverse Index */
16327                     do_escape(cons, c);
16328                     break;
16329                 default:
16330                     cons->c_esc_state = 0;
16331             }
16332         break;
16333
16334     case 2: /* ESC [ seen */
16335         if (c >= '0' && c <= '9') {
16336             if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
16337                 *cons->c_esc_parmp = *cons->c_esc_parmp * 10 + (c-'0');
16338         } else
16339         if (c == ';') {
16340             if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
16341                 cons->c_esc_parmp++;
16342         } else {
16343             do_escape(cons, c);
16344         }
16345     break;
16346 }
16347 }

16349 /*=====
16350 *          do_escape
16351 *=====
16352 PRIVATE void do_escape(cons, c)
16353 register console_t *cons; /* pointer to console struct */
16354 char c; /* next character in escape sequence */
16355 {
16356     int value, n;
16357     unsigned src, dst, count;
16358     int *parmp;
16359
16360     /* Some of these things hack on screen RAM, so it had better be up to date */
16361     flush(cons);
16362
16363     if (cons->c_esc_intro == '\0') {
16364         /* Handle a sequence beginning with just ESC */
16365         switch (c) {
16366             case 'M': /* Reverse Index */
16367                 if (cons->c_row == 0) {
16368                     scroll_screen(cons, SCROLL_DOWN);
16369                 } else {
16370                     cons->c_row--;
16371                 }
16372                 flush(cons);
16373                 break;
16374
16375             default: break;
16376         }
16377     } else
16378     if (cons->c_esc_intro == '[') {
16379         /* Handle a sequence beginning with ESC [ and parameters */

```

```

16380     value = cons->c_esc_parmv[0];
16381     switch (c) {
16382         case 'A':           /* ESC [nA moves up n lines */
16383             n = (value == 0 ? 1 : value);
16384             cons->c_row -= n;
16385             flush(cons);
16386             break;
16387
16388         case 'B':           /* ESC [nB moves down n lines */
16389             n = (value == 0 ? 1 : value);
16390             cons->c_row += n;
16391             flush(cons);
16392             break;
16393
16394         case 'C':           /* ESC [nC moves right n spaces */
16395             n = (value == 0 ? 1 : value);
16396             cons->c_column += n;
16397             flush(cons);
16398             break;
16399
16400         case 'D':           /* ESC [nD moves left n spaces */
16401             n = (value == 0 ? 1 : value);
16402             cons->c_column -= n;
16403             flush(cons);
16404             break;
16405
16406         case 'H':           /* ESC [m;nH" moves cursor to (m,n) */
16407             cons->c_row = cons->c_esc_parmv[0] - 1;
16408             cons->c_column = cons->c_esc_parmv[1] - 1;
16409             flush(cons);
16410             break;
16411
16412         case 'J':           /* ESC [sJ clears in display */
16413             switch (value) {
16414                 case 0:          /* Clear from cursor to end of screen */
16415                     count = scr_size - (cons->c_cur - cons->c_org);
16416                     dst = cons->c_cur;
16417                     break;
16418                 case 1:          /* Clear from start of screen to cursor */
16419                     count = cons->c_cur - cons->c_org;
16420                     dst = cons->c_org;
16421                     break;
16422                 case 2:          /* Clear entire screen */
16423                     count = scr_size;
16424                     dst = cons->c_org;
16425                     break;
16426                 default:        /* Do nothing */
16427                     count = 0;
16428                     dst = cons->c_org;
16429             }
16430             blank_color = cons->c_blank;
16431             mem_vid_copy(BLANK_MEM, dst, count);
16432             break;
16433
16434         case 'K':           /* ESC [sK clears line from cursor */
16435             switch (value) {
16436                 case 0:          /* Clear from cursor to end of line */
16437                     count = scr_width - cons->c_column;
16438                     dst = cons->c_cur;
16439                     break;
16440                 case 1:          /* Clear from beginning of line to cursor */
16441                     count = cons->c_column;
16442                     dst = cons->c_cur - cons->c_column;
16443                     break;
16444                 case 2:          /* Clear entire line */
16445                     count = scr_width;
16446                     dst = cons->c_cur - cons->c_column;

```

```

16447           break;
16448     default: /* Do nothing */
16449       count = 0;
16450       dst = cons->c_cur;
16451   }
16452   blank_color = cons->c_blank;
16453   mem_vid_copy(BLANK_MEM, dst, count);
16454   break;
16455
16456 case 'L':          /* ESC [nL inserts n lines at cursor */
16457   n = value;
16458   if (n < 1) n = 1;
16459   if (n > (scr_lines - cons->c_row))
16460     n = scr_lines - cons->c_row;
16461
16462   src = cons->c_org + cons->c_row * scr_width;
16463   dst = src + n * scr_width;
16464   count = (scr_lines - cons->c_row - n) * scr_width;
16465   vid_vid_copy(src, dst, count);
16466   blank_color = cons->c_blank;
16467   mem_vid_copy(BLANK_MEM, src, n * scr_width);
16468   break;
16469
16470 case 'M':          /* ESC [nM deletes n lines at cursor */
16471   n = value;
16472   if (n < 1) n = 1;
16473   if (n > (scr_lines - cons->c_row))
16474     n = scr_lines - cons->c_row;
16475
16476   dst = cons->c_org + cons->c_row * scr_width;
16477   src = dst + n * scr_width;
16478   count = (scr_lines - cons->c_row - n) * scr_width;
16479   vid_vid_copy(src, dst, count);
16480   blank_color = cons->c_blank;
16481   mem_vid_copy(BLANK_MEM, dst + count, n * scr_width);
16482   break;
16483
16484 case '@':          /* ESC [n@ inserts n chars at cursor */
16485   n = value;
16486   if (n < 1) n = 1;
16487   if (n > (scr_width - cons->c_column))
16488     n = scr_width - cons->c_column;
16489
16490   src = cons->c_cur;
16491   dst = src + n;
16492   count = scr_width - cons->c_column - n;
16493   vid_vid_copy(src, dst, count);
16494   blank_color = cons->c_blank;
16495   mem_vid_copy(BLANK_MEM, src, n);
16496   break;
16497
16498 case 'P':          /* ESC [nP deletes n chars at cursor */
16499   n = value;
16500   if (n < 1) n = 1;
16501   if (n > (scr_width - cons->c_column))
16502     n = scr_width - cons->c_column;
16503
16504   dst = cons->c_cur;
16505   src = dst + n;
16506   count = scr_width - cons->c_column - n;
16507   vid_vid_copy(src, dst, count);
16508   blank_color = cons->c_blank;
16509   mem_vid_copy(BLANK_MEM, dst + count, n);
16510   break;
16511
16512 case 'm':          /* ESC [nm enables rendition n */
16513   for (parmp = cons->c_esc_parmv; parmp <= cons->c_esc_parmp

```

```

16514             && parmp < bufend(cons->c_esc_parmv); parmp++) {
16515     if (cons->c_reverse) {
16516         /* Unswap fg and bg colors */
16517         cons->c_attr = ((cons->c_attr & 0x7000) >> 4) |
16518             ((cons->c_attr & 0x0700) << 4) |
16519             ((cons->c_attr & 0x8800));
16520     }
16521     switch (n = *parmp) {
16522         case 0: /* NORMAL */
16523             cons->c_attr = cons->c_blank = BLANK_COLOR;
16524             cons->c_reverse = FALSE;
16525             break;
16526
16527         case 1: /* BOLD */
16528             /* Set intensity bit */
16529             cons->c_attr |= 0x0800;
16530             break;
16531
16532         case 4: /* UNDERLINE */
16533             if (color) {
16534                 /* Change white to cyan, i.e. lose red */
16535                 /*
16536                 cons->c_attr = (cons->c_attr & 0xBBFF);
16537             } else {
16538                 /* Set underline attribute */
16539                 cons->c_attr = (cons->c_attr & 0x99FF);
16540             }
16541             break;
16542
16543         case 5: /* BLINKING */
16544             /* Set the blink bit */
16545             cons->c_attr |= 0x8000;
16546             break;
16547
16548         case 7: /* REVERSE */
16549             cons->c_reverse = TRUE;
16550             break;
16551
16552         default: /* COLOR */
16553             if (n == 39) n = 37; /* set default color */
16554             if (n == 49) n = 40;
16555
16556             if (!color) {
16557                 /* Don't mess up a monochrome screen */
16558             } else
16559             if (30 <= n && n <= 37) {
16560                 /* Foreground color */
16561                 cons->c_attr =
16562                     (cons->c_attr & 0xF8FF) |
16563                     (ansi_colors[(n - 30)] << 8);
16564                 cons->c_blank =
16565                     (cons->c_blank & 0xF8FF) |
16566                     (ansi_colors[(n - 30)] << 8);
16567             } else
16568             if (40 <= n && n <= 47) {
16569                 /* Background color */
16570                 cons->c_attr =
16571                     (cons->c_attr & 0x8FFF) |
16572                     (ansi_colors[(n - 40)] << 12);
16573                 cons->c_blank =
16574                     (cons->c_blank & 0x8FFF) |
16575                     (ansi_colors[(n - 40)] << 12);
16576             }
16577         }
16578     if (cons->c_reverse) {
16579         /* Swap fg and bg colors */

```

```

16580                               cons->c_attr = ((cons->c_attr & 0x7000) >> 4) |
16581                               (((cons->c_attr & 0x0700) << 4) |
16582                               ((cons->c_attr & 0x8800)));
16583                         }
16584                     }
16585                 break;
16586             }
16587         }
16588     cons->c_esc_state = 0;
16589 }

16591 /*=====
16592 *          set_6845
16593 *=====
16594 PRIVATE void set_6845(reg, val)
16595 int reg;           /* which register pair to set */
16596 unsigned val;      /* 16-bit value to set it to */
16597 {
16598 /* Set a register pair inside the 6845.
16599 * Registers 12-13 tell the 6845 where in video ram to start
16600 * Registers 14-15 tell the 6845 where to put the cursor
16601 */
16602     pvb_pair_t char_out[4];
16603     pv_set(char_out[0], vid_port + INDEX, reg);    /* set index register */
16604     pv_set(char_out[1], vid_port + DATA, (val>>8) & BYTE);    /* high byte */
16605     pv_set(char_out[2], vid_port + INDEX, reg + 1);        /* again */
16606     pv_set(char_out[3], vid_port + DATA, val&BYTE);        /* low byte */
16607     sys_voutb(char_out, 4);                            /* do actual output */
16608 }

16610 /*=====
16611 *          get_6845
16612 *=====
16613 PRIVATE void get_6845(reg, val)
16614 int reg;           /* which register pair to set */
16615 unsigned *val;      /* 16-bit value to set it to */
16616 {
16617     char v1, v2;
16618 /* Get a register pair inside the 6845. */
16619     sys_outb(vid_port + INDEX, reg);
16620     sys_inb(vid_port + DATA, &v1);
16621     sys_outb(vid_port + INDEX, reg+1);
16622     sys_inb(vid_port + DATA, &v2);
16623     *val = (v1 << 8) | v2;
16624 }

16626 /*=====
16627 *          beep
16628 *=====
16629 PRIVATE void beep()
16630 {
16631 /* Making a beeping sound on the speaker (output for CRTL-G).
16632 * This routine works by turning on the bits 0 and 1 in port B of the 8255
16633 * chip that drive the speaker.
16634 */
16635     static timer_t tmr_stop_beep;
16636     pvb_pair_t char_out[3];
16637     clock_t now;
16638     int port_b_val, s;
16639
16640 /* Fetch current time in advance to prevent beeping delay. */
16641 if ((s=getuptime(&now)) != OK)
16642     panic("TTY","Console couldn't get clock's uptime.", s);
16643 if (!beeping) {
16644     /* Set timer channel 2, square wave, with given frequency. */
16645     pv_set(char_out[0], TIMER_MODE, 0xB6);
16646     pv_set(char_out[1], TIMER2, (BEEP_FREQ >> 0) & BYTE);

```

```

16647      pv_set(char_out[2], TIMER2, (BEEP_FREQ >> 8) & BYTE);
16648      if (sys_voutb(char_out, 3)==OK) {
16649          if (sys_inb(PORT_B, &port_b_val)==OK &&
16650              sys_outb(PORT_B, (port_b_val|3))==OK)
16651              beeping = TRUE;
16652      }
16653  }
16654  /* Add a timer to the timers list. Possibly reschedule the alarm. */
16655  tmrs_settimer(&tty_timers, &tmr_stop_beep, now+B_TIME, stop_beep, NULL);
16656  if (tty_timers->tmr_exp_time != tty_next_timeout) {
16657      tty_next_timeout = tty_timers->tmr_exp_time;
16658      if ((s=sys_setalarm(tty_next_timeout, 1)) != OK)
16659          panic("TTY","Console couldn't set alarm.", s);
16660  }
16661 }

16663 /*=====
16664 *           stop_beep
16665 *=====
16666 PRIVATE void stop_beep(tmrp)
16667 timer_t *tmrp;
16668 {
16669 /* Turn off the beeper by turning off bits 0 and 1 in PORT_B. */
16670     int port_b_val;
16671     if (sys_inb(PORT_B, &port_b_val)==OK &&
16672         sys_outb(PORT_B, (port_b_val & ~3))==OK)
16673         beeping = FALSE;
16674 }

16676 /*=====
16677 *           scr_init
16678 *=====
16679 PUBLIC void scr_init(tp)
16680 tty_t *tp;
16681 {
16682 /* Initialize the screen driver. */
16683     console_t *cons;
16684     phys_bytes vid_base;
16685     u16_t bios_columns, bios_crtbase, bios_fontlines;
16686     u8_t bios_rows;
16687     int line;
16688     int s;
16689     static int vdu_initialized = 0;
16690     unsigned page_size;
16691
16692 /* Associate console and TTY. */
16693     line = tp - &tty_table[0];
16694     if (line >= nr_cons) return;
16695     cons = &cons_table[line];
16696     cons->c_tty = tp;
16697     tp->tty_priv = cons;
16698
16699 /* Initialize the keyboard driver. */
16700     kb_init(tp);
16701
16702 /* Fill in TTY function hooks. */
16703     tp->tty_devwrite = cons_write;
16704     tp->tty_echo = cons_echo;
16705     tp->tty_ioctl = cons_ioctl;
16706
16707 /* Get the BIOS parameters that describe the VDU. */
16708     if (!vdu_initialized++) {
16709
16710         /* How about error checking? What to do on failure??? */
16711         s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_SCREEN_COLS_ADDR,
16712                         SELF, D, (vir_bytes) &bios_columns, VDU_SCREEN_COLS_SIZE);
16713         s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_CRT_BASE_ADDR,

```

```

16714             SELF, D, (vir_bytes) &bios_crtbase, VDU_CRT_BASE_SIZE);
16715             s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_SCREEN_ROWS_ADDR,
16716                         SELF, D, (vir_bytes) &bios_rows, VDU_SCREEN_ROWS_SIZE);
16717             s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_FONTLINES_ADDR,
16718                         SELF, D, (vir_bytes) &bios_fontlines, VDU_FONTLINES_SIZE);
16719
16720             vid_port = bios_crtbase;
16721             scr_width = bios_columns;
16722             font_lines = bios_fontlines;
16723             scr_lines = machine.vdu_ega ? bios_rows+1 : 25;
16724
16725             if (color) {
16726                 vid_base = COLOR_BASE;
16727                 vid_size = COLOR_SIZE;
16728             } else {
16729                 vid_base = MONO_BASE;
16730                 vid_size = MONO_SIZE;
16731             }
16732             if (machine.vdu_ega) vid_size = EGA_SIZE;
16733             wrap = ! machine.vdu_ega;
16734
16735             s = sys_segctl(&vid_index, &vid_seg, &vid_off, vid_base, vid_size);
16736
16737             vid_size >= 1;           /* word count */
16738             vid_mask = vid_size - 1;
16739
16740             /* Size of the screen (number of displayed characters.) */
16741             scr_size = scr_lines * scr_width;
16742
16743             /* There can be as many consoles as video memory allows. */
16744             nr_cons = vid_size / scr_size;
16745             if (nr_cons > NR_CONS) nr_cons = NR_CONS;
16746             if (nr_cons > 1) wrap = 0;
16747             page_size = vid_size / nr_cons;
16748         }
16749
16750         cons->c_start = line * page_size;
16751         cons->c_limit = cons->c_start + page_size;
16752         cons->c_cur = cons->c_org = cons->c_start;
16753         cons->c_attr = cons->c_blank = BLANK_COLOR;
16754
16755         if (line != 0) {
16756             /* Clear the non-console vtys. */
16757             blank_color = BLANK_COLOR;
16758             mem_vid_copy(BLANK_MEM, cons->c_start, scr_size);
16759         } else {
16760             int i, n;
16761             /* Set the cursor of the console vty at the bottom. c_cur
16762             * is updated automatically later.
16763             */
16764             scroll_screen(cons, SCROLL_UP);
16765             cons->c_row = scr_lines - 1;
16766             cons->c_column = 0;
16767         }
16768         select_console(0);
16769         cons_ioctl(tp, 0);
16770     }
16771
16772     /*=====
16773     *                               kputc
16774     *=====
16775 PUBLIC void kputc(c)
16776 int c;
16777 {
16778     putk(c);
16779 }
```

```

16781 /*=====
16782 *          do_new_kmess
16783 *=====
16784 PUBLIC void do_new_kmess(m)
16785 message *m;
16786 {
16787 /* Notification for a new kernel message. */
16788     struct kmessages kmess;           /* kmessages structure */
16789     static int prev_next = 0;         /* previous next seen */
16790     int size, next;
16791     int bytes;
16792     int r;
16793
16794     /* Try to get a fresh copy of the buffer with kernel messages. */
16795     sys_getkmessages(&kmess);
16796
16797     /* Print only the new part. Determine how many new bytes there are with
16798      * help of the current and previous 'next' index. Note that the kernel
16799      * buffer is circular. This works fine if less than KMESS_BUF_SIZE bytes
16800      * is new data; else we miss % KMESS_BUF_SIZE here.
16801      * Check for size being positive, the buffer might as well be emptied!
16802      */
16803     if (kmess.km_size > 0) {
16804         bytes = ((kmess.km_next + KMESS_BUF_SIZE) - prev_next) % KMESS_BUF_SIZE;
16805         r=prev_next;                  /* start at previous old */
16806         while (bytes > 0) {
16807             putk( kmess.km_buf[(r%KMESS_BUF_SIZE)] );
16808             bytes--;
16809             r++;
16810         }
16811         putk(0);                      /* terminate to flush output */
16812     }
16813
16814     /* Almost done, store 'next' so that we can determine what part of the
16815      * kernel messages buffer to print next time a notification arrives.
16816      */
16817     prev_next = kmess.km_next;
16818 }

16820 /*=====
16821 *          do_diagnostics
16822 *=====
16823 PUBLIC void do_diagnostics(m_ptr)
16824 message *m_ptr;                  /* pointer to request message */
16825 {
16826     /* Print a string for a server. */
16827     char c;
16828     vir_bytes src;
16829     int count;
16830     int result = OK;
16831     int proc_nr = m_ptr->DIAG_PROC_NR;
16832     if (proc_nr == SELF) proc_nr = m_ptr->m_source;
16833
16834     src = (vir_bytes) m_ptr->DIAG_PRINT_BUF;
16835     for (count = m_ptr->DIAG_BUF_COUNT; count > 0; count--) {
16836         if (sys_vircopy(proc_nr, D, src++, SELF, D, (vir_bytes) &c, 1) != OK) {
16837             result = EFAULT;
16838             break;
16839         }
16840         putk(c);
16841     }
16842     putk(0);                      /* always terminate, even with EFAULT */
16843     m_ptr->m_type = result;
16844     send(m_ptr->m_source, m_ptr);
16845 }

```

```

16847 /*=====
16848 *          putk
16849 *=====
16850 PRIVATE void putk(c)
16851 int c;           /* character to print */
16852 {
16853 /* This procedure is used by the version of printf() that is linked with
16854 * the TTY driver. The one in the library sends a message to FS, which is
16855 * not what is needed for printing within the TTY. This version just queues
16856 * the character and starts the output.
16857 */
16858 if (c != 0) {
16859     if (c == '\n') putk('\r');
16860     out_char(&cons_table[0], (int) c);
16861 } else {
16862     flush(&cons_table[0]);
16863 }
16864 }

16865 /*=====
16866 *          toggle_scroll
16867 *=====
16868 PUBLIC void toggle_scroll()
16869 {
16870 /* Toggle between hardware and software scroll. */
16871
16872     cons_org0();
16873     softscroll = !softscroll;
16874     printf("%sware scrolling enabled.\n", softscroll ? "Soft" : "Hard");
16875 }
16876 */

16877 /*=====
16878 *          cons_stop
16879 *=====
16880 PUBLIC void cons_stop()
16881 {
16882 /* Prepare for halt or reboot. */
16883     cons_org0();
16884     softscroll = 1;
16885     select_console(0);
16886     cons_table[0].c_attr = cons_table[0].c_blank = BLANK_COLOR;
16887 }
16888 */

16889 /*=====
16900 *          cons_org0
16901 *=====
16902 PRIVATE void cons_org0()
16903 {
16904 /* Scroll video memory back to put the origin at 0. */
16905     int cons_line;
16906     console_t *cons;
16907     unsigned n;
16908
16909     for (cons_line = 0; cons_line < nr_cons; cons_line++) {
16910         cons = &cons_table[cons_line];
16911         while (cons->c_org > cons->c_start) {
16912             n = vid_size - scr_size;           /* amount of unused memory */
16913             if (n > cons->c_org - cons->c_start)
16914                 n = cons->c_org - cons->c_start;
16915             vid_vid_copy(cons->c_org, cons->c_org - n, scr_size);
16916             cons->c_org -= n;
16917         }
16918         flush(cons);
16919     }
16920     select_console(ccurrent);
16921 }
16922 }
```

```

16914 /*=====
16915 *          select_console
16916 *=====
16917 PUBLIC void select_console(int cons_line)
16918 {
16919 /* Set the current console to console number 'cons_line'. */
16920
16921     if (cons_line < 0 || cons_line >= nr_cons) return;
16922     ccurrent = cons_line;
16923     curcons = &cons_table[cons_line];
16924     set_6845(VID_ORG, curcons->c_org);
16925     set_6845(CURSOR, curcons->c_cur);
16926 }

16927 /*=====
16928 *          con_loadfont
16929 *=====
16930 PUBLIC int con_loadfont(m)
16931 message *m;
16932 {
16933 /* Load a font into the EGA or VGA adapter. */
16934     int result;
16935     static struct sequence seq1[7] = {
16936         { GA_SEQUENCER_INDEX, 0x00, 0x01 },
16937         { GA_SEQUENCER_INDEX, 0x02, 0x04 },
16938         { GA_SEQUENCER_INDEX, 0x04, 0x07 },
16939         { GA_SEQUENCER_INDEX, 0x00, 0x03 },
16940         { GA_GRAPHICS_INDEX, 0x04, 0x02 },
16941         { GA_GRAPHICS_INDEX, 0x05, 0x00 },
16942         { GA_GRAPHICS_INDEX, 0x06, 0x00 }.
16943     };
16944     static struct sequence seq2[7] = {
16945         { GA_SEQUENCER_INDEX, 0x00, 0x01 },
16946         { GA_SEQUENCER_INDEX, 0x02, 0x03 },
16947         { GA_SEQUENCER_INDEX, 0x04, 0x03 },
16948         { GA_SEQUENCER_INDEX, 0x00, 0x03 },
16949         { GA_GRAPHICS_INDEX, 0x04, 0x00 },
16950         { GA_GRAPHICS_INDEX, 0x05, 0x10 },
16951         { GA_GRAPHICS_INDEX, 0x06, 0 } ,
16952     };
16953
16954     seq2[6].value= color ? 0x0E : 0x0A;
16955
16956     if (!machine.vdu_egga) return(ENOTTY);
16957     result = ga_program(seq1); /* bring font memory into view */
16958
16959     result = sys_physcopy(m->PROC_NR, D, (vir_bytes) m->ADDRESS,
16960                           NONE, PHYS_SEG, (phys_bytes) GA_VIDEO_ADDRESS, (phys_bytes)GA_FONT_SIZE);
16961
16962     result = ga_program(seq2); /* restore */
16963
16964     return(result);
16965 }
16966 }

16967 /*=====
16968 *          ga_program
16969 *=====
16970 PRIVATE int ga_program(seq)
16971 struct sequence *seq;
16972 {
16973     pvb_pair_t char_out[14];
16974     int i;
16975     for (i=0; i<7; i++) {
16976         pv_set(char_out[2*i], seq->index, seq->port);
16977         pv_set(char_out[2*i+1], seq->index+1, seq->value);
16978         seq++;
16979     }

```

```

16981     return sys_voutb(char_out, 14);
16982 }

16984 /*=====
16985 *          cons_ioctl
16986 *=====
16987 PRIVATE int cons_ioctl(tp, try)
16988 tty_t *tp;
16989 int try;
16990 {
16991 /* Set the screen dimensions. */
16992
16993     tp->tty_winsize.ws_row= scr_lines;
16994     tp->tty_winsize.ws_col= scr_width;
16995     tp->tty_winsize.ws_xpixel= scr_width * 8;
16996     tp->tty_winsize.ws_ypixel= scr_lines * font_lines;
16997 }

+++++
servers/pm/pm.h
+++++

17000 /* This is the master header for PM. It includes some other files
17001 * and defines the principal constants.
17002 */
17003 #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
17004 #define _MINIX             1    /* tell headers to include MINIX stuff */
17005 #define _SYSTEM            1    /* tell headers that this is the kernel */
17006
17007 /* The following are so basic, all the *.c files get them automatically. */
17008 #include <minix/config.h>      /* MUST be first */
17009 #include <ansi.h>           /* MUST be second */
17010 #include <sys/types.h>
17011 #include <minix/const.h>
17012 #include <minix/type.h>
17013
17014 #include <fcntl.h>
17015 #include <unistd.h>
17016 #include <minix/syslib.h>
17017 #include <minix/sysutil.h>
17018
17019 #include <limits.h>
17020 #include <errno.h>
17021
17022 #include "const.h"
17023 #include "type.h"
17024 #include "proto.h"
17025 #include "glo.h"

+++++
servers/pm/const.h
+++++

17100 /* Constants used by the Process Manager. */
17101
17102 #define NO_MEM ((phys_clicks) 0) /* returned by alloc_mem() with mem is up */
17103
17104 #define NR_PIDS      30000   /* process ids range from 0 to NR_PIDS-1.
17105 * (magic constant: some old applications use
17106 * a 'short' instead of pid_t.)
```

```

17107                                     */
17108
17109 #define PM_PID          0 /* PM's process id number */
17110 #define INIT_PID        1 /* INIT's process id number */
17111

+++++servers/pm/type.h+++++
17200 /* If there were any type definitions local to the Process Manager, they would
17201 * be here. This file is included only for symmetry with the kernel and File
17202 * System, which do have some local type definitions.
17203 */
17204

+++++servers/pm/proto.h+++++
17300 /* Function prototypes. */
17301
17302 struct mproc;
17303 struct stat;
17304 struct mem_map;
17305 struct memory;
17306
17307 #include <timers.h>
17308
17309 /* alloc.c */
17310 _PROTOTYPE( phys_ticks alloc_mem, (phys_ticks clicks) );
17311 _PROTOTYPE( void free_mem, (phys_ticks base, phys_ticks clicks) );
17312 _PROTOTYPE( void mem_init, (struct memory *chunks, phys_ticks *free) );
17313 #define swap_in()           ((void)0)
17314 #define swap_inqueue(rmp)    ((void)0)
17315
17316 /* break.c */
17317 _PROTOTYPE( int adjust, (struct mproc *rmp,
17318             vir_ticks data_ticks, vir_bytes sp) );
17319 _PROTOTYPE( int do_brk, (void) );
17320 _PROTOTYPE( int size_ok, (int file_type, vir_ticks tc, vir_ticks dc,
17321             vir_ticks sc, vir_ticks dvir, vir_ticks s_vir) );
17322
17323 /* devio.c */
17324 _PROTOTYPE( int do_dev_io, (void) );
17325 _PROTOTYPE( int do_dev_io, (void) );
17326
17327 /* dmp.c */
17328 _PROTOTYPE( int do_fkey_pressed, (void) );
17329
17330 /* exec.c */
17331 _PROTOTYPE( int do_exec, (void) );
17332 _PROTOTYPE( void rw_seg, (int rw, int fd, int proc, int seg,
17333             phys_bytes seg_bytes) );
17334 _PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
17335             Dev_t dev, time_t ctime) );
17336
17337 /* forkexit.c */
17338 _PROTOTYPE( int do_fork, (void) );
17339 _PROTOTYPE( int do_pm_exit, (void) );
17340 _PROTOTYPE( int do_waitpid, (void) );
17341 _PROTOTYPE( void pm_exit, (struct mproc *rmp, int exit_status) );
17342
17343 /* getset.c */
17344 _PROTOTYPE( int do_getset, (void) );

```

```

17345 /* main.c */
17346 _PROTOTYPE( int main, (void) );
17348
17349 /* misc.c */
17350 _PROTOTYPE( int do_reboot, (void) );
17351 _PROTOTYPE( int do_getsysinfo, (void) );
17352 _PROTOTYPE( int do_getprocnr, (void) );
17353 _PROTOTYPE( int do_svrcntl, (void) );
17354 _PROTOTYPE( int do_allocmem, (void) );
17355 _PROTOTYPE( int do_freemem, (void) );
17356 _PROTOTYPE( int do_getsetpriority, (void) );
17357
17358 _PROTOTYPE( void setreply, (int proc_nr, int result) );
17359
17360 /* signal.c */
17361 _PROTOTYPE( int do_alarm, (void) );
17362 _PROTOTYPE( int do_kill, (void) );
17363 _PROTOTYPE( int ksig_pending, (void) );
17364 _PROTOTYPE( int do_pause, (void) );
17365 _PROTOTYPE( int set_alarm, (int proc_nr, int sec) );
17366 _PROTOTYPE( int check_sig, (pid_t proc_id, int signo) );
17367 _PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr) );
17368 _PROTOTYPE( int do_sigaction, (void) );
17369 _PROTOTYPE( int do_sigpending, (void) );
17370 _PROTOTYPE( int do_sigprocmask, (void) );
17371 _PROTOTYPE( int do_sigreturn, (void) );
17372 _PROTOTYPE( int do_sigsuspend, (void) );
17373 _PROTOTYPE( void check_pending, (struct mproc *rmp) );
17374
17375 /* time.c */
17376 _PROTOTYPE( int do_stime, (void) );
17377 _PROTOTYPE( int do_time, (void) );
17378 _PROTOTYPE( int do_times, (void) );
17379 _PROTOTYPE( int do_gettimeofday, (void) );
17380
17381 /* timers.c */
17382 _PROTOTYPE( void pm_set_timer, (timer_t *tp, int delta,
17383             tmr_func_t watchdog, int arg) );
17384 _PROTOTYPE( void pm_expire_timers, (clock_t now) );
17385 _PROTOTYPE( void pm_cancel_timer, (timer_t *tp) );
17386
17387 /* trace.c */
17388 _PROTOTYPE( int do_trace, (void) );
17389 _PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr) );
17390
17391 /* utility.c */
17392 _PROTOTYPE( pid_t get_free_pid, (void) );
17393 _PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
17394 _PROTOTYPE( int no_sys, (void) );
17395 _PROTOTYPE( void panic, (char *who, char *mess, int num) );
17396 _PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3) );
17397 _PROTOTYPE( int get_stack_ptr, (int proc_nr, vir_bytes *sp) );
17398 _PROTOTYPE( int get_mem_map, (int proc_nr, struct mem_map *mem_map) );
17399 _PROTOTYPE( char *find_param, (const char *key) );
17400 _PROTOTYPE( int proc_from_pid, (pid_t p) );

+++++
servers/pm/glo.h
+++++
17500 /* EXTERN should be extern except in table.c */
17501 #ifdef _TABLE
17502 #undef EXTERN
17503 #define EXTERN
17504 #endif

```

```

17505
17506 /* Global variables. */
17507 EXTERN struct mproc *mp;           /* ptr to 'mproc' slot of current process */
17508 EXTERN int procs_in_use;          /* how many processes are marked as IN_USE */
17509 EXTERN char monitor_params[128*sizeof(char *)]; /* boot monitor parameters */
17510 EXTERN struct kinfo kinfo;        /* kernel information */

17511
17512 /* The parameters of the call are kept here. */
17513 EXTERN message m_in;             /* the incoming message itself is kept here. */
17514 EXTERN int who;                /* caller's proc number */
17515 EXTERN int call_nr;             /* system call number */

17516
17517 extern _PROTOTYPE (int (*call_vec[]), (void) ); /* system call handlers */
17518 extern char core_name[];         /* file name where core images are produced */
17519 EXTERN sigset_t core_sset;       /* which signals cause core images */
17520 EXTERN sigset_t ign_sset;        /* which signals are by default ignored */
17521

```

```
+++++
servers/pm/mproc.h
+++++
```

```

17600 /* This table has one slot per process. It contains all the process management
17601 * information for each process. Among other things, it defines the text, data
17602 * and stack segments, uids and gids, and various flags. The kernel and file
17603 * systems have tables that are also indexed by process, with the contents
17604 * of corresponding slots referring to the same process in all three.
17605 */
17606 #include <timers.h>
17607
17608 EXTERN struct mproc {
17609     struct mem_map mp_seg[NR_LOCAL_SEGS]; /* points to text, data, stack */
17610     char mp_exitstatus;                  /* storage for status when process exits */
17611     char mp_sigstatus;                 /* storage for signal # for killed procs */
17612     pid_t mp_pid;                     /* process id */
17613     pid_t mp_progrp;                 /* pid of process group (used for signals) */
17614     pid_t mp_wpid;                   /* pid this process is waiting for */
17615     int mp_parent;                   /* index of parent process */
17616
17617 /* Child user and system times. Accounting done on child exit. */
17618 clock_t mp_child_utime;            /* cumulative user time of children */
17619 clock_t mp_child_stime;            /* cumulative sys time of children */
17620
17621 /* Real and effective uids and gids. */
17622 uid_t mp_realmuid;               /* process' real uid */
17623 uid_t mp_effuid;                /* process' effective uid */
17624 gid_t mp_realgid;              /* process' real gid */
17625 gid_t mp_effgid;               /* process' effective gid */
17626
17627 /* File identification for sharing. */
17628 ino_t mp_ino;                  /* inode number of file */
17629 dev_t mp_dev;                  /* device number of file system */
17630 time_t mp_ctime;                /* inode changed time */
17631
17632 /* Signal handling information. */
17633 sigset_t mp_ignore;             /* 1 means ignore the signal, 0 means don't */
17634 sigset_t mp_catch;              /* 1 means catch the signal, 0 means don't */
17635 sigset_t mp_sig2mess;            /* 1 means transform into notify message */
17636 sigset_t mp_sigmask;             /* signals to be blocked */
17637 sigset_t mp_sigmask2;            /* saved copy of mp_sigmask */
17638 sigset_t mp_sigpending;          /* pending signals to be handled */
17639 struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
17640 vir_bytes mp_sigreturn;          /* address of C library __sigreturn function */
17641 struct timer mp_timer;           /* watchdog timer for alarm(2) */
17642

```

```

17643 /* Backwards compatibility for signals. */
17644 sighandler_t mp_func;           /* all sigs vectored to a single user fcn */
17645
17646     unsigned mp_flags;          /* flag bits */
17647     vir_bytes mp_prcargs;       /* ptr to proc's initial stack arguments */
17648     struct mproc *mp_swapq;     /* queue of procs waiting to be swapped in */
17649     message mp_reply;         /* reply message to be sent to one */
17650
17651 /* Scheduling priority. */
17652     signed int mp_nice;        /* nice is PRIO_MIN..PRIO_MAX, standard 0. */
17653
17654     char mp_name[PROC_NAME_LEN]; /* process name */
17655 } mproc[NR_PROCS];
17656
17657 /* Flag values */
17658 #define IN_USE          0x001 /* set when 'mproc' slot in use */
17659 #define WAITING         0x002 /* set by WAIT system call */
17660 #define ZOMBIE          0x004 /* set by EXIT, cleared by WAIT */
17661 #define PAUSED           0x008 /* set by PAUSE system call */
17662 #define ALARM_ON        0x010 /* set when SIGALRM timer started */
17663 #define SEPARATE         0x020 /* set if file is separate I & D space */
17664 #define TRACED           0x040 /* set if process is to be traced */
17665 #define STOPPED          0x080 /* set if process stopped for tracing */
17666 #define SIGSUSPENDED    0x100 /* set by SIGSUSPEND system call */
17667 #define REPLY             0x200 /* set if a reply message is pending */
17668 #define ONSWAP            0x400 /* set if data segment is swapped out */
17669 #define SWAPIN            0x800 /* set if on the "swap this in" queue */
17670 #define DONT_SWAP         0x1000 /* never swap out this process */
17671 #define PRIV_PROC         0x2000 /* system process, special privileges */
17672
17673 #define NIL_MPROC ((struct mproc *) 0)
17674

+++++
servers/pm/param.h
+++++
17700 /* The following names are synonyms for the variables in the input message. */
17701 #define addr             m1_pl
17702 #define exec_name        m1_pl
17703 #define exec_len          m1_il
17704 #define func              m6_f1
17705 #define grp_id           m1_il
17706 #define namelen          m1_i2
17707 #define pid               m1_il
17708 #define procnr           m1_il
17709 #define seconds          m1_il
17710 #define sig                m6_il
17711 #define stack_bytes       m1_i2
17712 #define stack_ptr         m1_p2
17713 #define status            m1_i1
17714 #define usr_id            m1_il
17715 #define request           m2_i2
17716 #define taddr              m2_ll
17717 #define data              m2_l2
17718 #define sig_nr            m1_i2
17719 #define sig_nsa           m1_p1
17720 #define sig_osa            m1_p2
17721 #define sig_ret           m1_p3
17722 #define sig_set            m2_ll
17723 #define sig_how            m2_il
17724 #define sig_flags          m2_i2
17725 #define sig_context        m2_p1
17726 #define info_what          m1_il
17727 #define info_where         m1_pl
17728 #define reboot_flag        m1_il
17729 #define reboot_code        m1_pl

```

```

17730 #define reboot_strlen    m1_i2
17731 #define svrctl_req      m2_i1
17732 #define svrctl_argp     m2_p1
17733 #define stime           m2_l1
17734 #define memsize          m4_l1
17735 #define membase          m4_l2
17736
17737 /* The following names are synonyms for the variables in a reply message. */
17738 #define reply_res        m_type
17739 #define reply_res2       m2_i1
17740 #define reply_ptr        m2_p1
17741 #define reply_mask       m2_l1
17742 #define reply_trace      m2_l2
17743 #define reply_time       m2_l1
17744 #define reply_utime      m2_l2
17745 #define reply_t1         m4_l1
17746 #define reply_t2         m4_l2
17747 #define reply_t3         m4_l3
17748 #define reply_t4         m4_l4
17749 #define reply_t5         m4_l5
17750
17751 /* The following names are used to inform the FS about certain events. */
17752 #define tell_fs_arg1      m1_i1
17753 #define tell_fs_arg2      m1_i2
17754 #define tell_fs_arg3      m1_i3
17755

+++++servers/pm/table.c+++++
17800 /* This file contains the table used to map system call numbers onto the
17801   * routines that perform them.
17802   */
17803
17804 #define _TABLE
17805
17806 #include "pm.h"
17807 #include <minix/callnr.h>
17808 #include <signal.h>
17809 #include "mproc.h"
17810 #include "param.h"
17811
17812 /* Miscellaneous */
17813 char core_name[] = "core";      /* file name where core images are produced */
17814
17815 _PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
17816     no_sys,           /* 0 = unused */
17817     do_pm_exit,       /* 1 = exit */
17818     do_fork,          /* 2 = fork */
17819     no_sys,           /* 3 = read */
17820     no_sys,           /* 4 = write */
17821     no_sys,           /* 5 = open */
17822     no_sys,           /* 6 = close */
17823     do_waitpid,      /* 7 = wait */
17824     no_sys,           /* 8 = creat */
17825     no_sys,           /* 9 = link */
17826     no_sys,           /* 10 = unlink */
17827     do_waitpid,      /* 11 = waitpid */
17828     no_sys,           /* 12 = chdir */
17829     do_time,          /* 13 = time */
17830     no_sys,           /* 14 = mknod */
17831     no_sys,           /* 15 = chmod */
17832     no_sys,           /* 16 = chown */
17833     do_brk,           /* 17 = break */
17834     no_sys,           /* 18 = stat */

```

```

17835      no_sys,          /* 19 = lseek   */
17836      do_getset,       /* 20 = getpid */
17837      no_sys,          /* 21 = mount   */
17838      no_sys,          /* 22 = umount */
17839      do_getset,       /* 23 = setuid */
17840      do_getset,       /* 24 = getuid */
17841      do_stime,         /* 25 = stime   */
17842      do_trace,         /* 26 = ptrace */
17843      do_alarm,         /* 27 = alarm   */
17844      no_sys,          /* 28 = fstat   */
17845      do_pause,         /* 29 = pause   */
17846      no_sys,          /* 30 = utime   */
17847      no_sys,          /* 31 = (stty) */
17848      no_sys,          /* 32 = (gtty) */
17849      no_sys,          /* 33 = access */
17850      no_sys,          /* 34 = (nice) */
17851      no_sys,          /* 35 = (ftime) */
17852      no_sys,          /* 36 = sync   */
17853      do_kill,          /* 37 = kill   */
17854      no_sys,          /* 38 = rename */
17855      no_sys,          /* 39 = mkdir   */
17856      no_sys,          /* 40 = rmdir   */
17857      no_sys,          /* 41 = dup   */
17858      no_sys,          /* 42 = pipe   */
17859      do_times,         /* 43 = times */
17860      no_sys,          /* 44 = (prof) */
17861      no_sys,          /* 45 = unused */
17862      do_getset,       /* 46 = setgid */
17863      do_getset,       /* 47 = getgid */
17864      no_sys,          /* 48 = (signal) */
17865      no_sys,          /* 49 = unused */
17866      no_sys,          /* 50 = unused */
17867      no_sys,          /* 51 = (acct) */
17868      no_sys,          /* 52 = (phys) */
17869      no_sys,          /* 53 = (lock) */
17870      no_sys,          /* 54 = ioctl   */
17871      no_sys,          /* 55 = fcntl   */
17872      no_sys,          /* 56 = (mpx) */
17873      no_sys,          /* 57 = unused */
17874      no_sys,          /* 58 = unused */
17875      do_exec,          /* 59 = execve */
17876      no_sys,          /* 60 = umask   */
17877      no_sys,          /* 61 = chroot */
17878      do_getset,       /* 62 = setsid */
17879      do_getset,       /* 63 = getpgrp */
17880
17881      no_sys,          /* 64 = unused */
17882      no_sys,          /* 65 = UNPAUSE */
17883      no_sys,          /* 66 = unused */
17884      no_sys,          /* 67 = REVIVE */
17885      no_sys,          /* 68 = TASK_REPLY */
17886      no_sys,          /* 69 = unused */
17887      no_sys,          /* 70 = unused */
17888      do_sigaction,    /* 71 = sigaction */
17889      do_sigsuspend,   /* 72 = sigsuspend */
17890      do_sigpending,   /* 73 = sigpending */
17891      do_sigprocmask,  /* 74 = sigprocmask */
17892      do_sigreturn,    /* 75 = sigreturn */
17893      do_reboot,       /* 76 = reboot */
17894      do_svrcntl,     /* 77 = svrcntl */
17895
17896      no_sys,          /* 78 = unused */
17897      do_getsysinfo,   /* 79 = getsysinfo */
17898      do_getprocnr,    /* 80 = getprocnr */
17899      no_sys,          /* 81 = unused */
17900      no_sys,          /* 82 = fstatfs */
17901      do_allocmem,    /* 83 = memalloc */

```

```

17902     do_freemem,      /* 84 = memfree */
17903     no_sys,          /* 85 = select */
17904     no_sys,          /* 86 = fchdir */
17905     no_sys,          /* 87 = fsync */
17906     do_getsetpriority, /* 88 = getpriority */
17907     do_getsetpriority, /* 89 = setpriority */
17908     do_time,          /* 90 = gettimeofday */
17909 };
17910 /* This should not fail with "array size is negative": */
17911 extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];

+++++
servers/pm/main.c
++++

18000 /* This file contains the main program of the process manager and some related
18001 * procedures. When MINIX starts up, the kernel runs for a little while,
18002 * initializing itself and its tasks, and then it runs PM and FS. Both PM
18003 * and FS initialize themselves as far as they can. PM asks the kernel for
18004 * all free memory and starts serving requests.
18005 *
18006 * The entry points into this file are:
18007 *   main:      starts PM running
18008 *   setreply:  set the reply to be sent to process making an PM system call
18009 */
18010
18011 #include "pm.h"
18012 #include <minix/keymap.h>
18013 #include <minix/callnr.h>
18014 #include <minix/com.h>
18015 #include <signal.h>
18016 #include <stdlib.h>
18017 #include <fcntl.h>
18018 #include <sys/resource.h>
18019 #include <string.h>
18020 #include "mproc.h"
18021 #include "param.h"
18022
18023 #include "../../kernel/const.h"
18024 #include "../../kernel/config.h"
18025 #include "../../kernel/type.h"
18026 #include "../../kernel/proc.h"
18027
18028 FORWARD _PROTOTYPE( void get_work, (void) );;
18029 FORWARD _PROTOTYPE( void pm_init, (void) );
18030 FORWARD _PROTOTYPE( int get_nice_value, (int queue) );
18031 FORWARD _PROTOTYPE( void get_mem_chunks, (struct memory *mem_chunks) );
18032 FORWARD _PROTOTYPE( void patch_mem_chunks, (struct memory *mem_chunks,
18033           struct mem_map *map_ptr) );
18034
18035 #define click_to_round_k(n) \
18036     ((unsigned) (((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024)
18037
18038 /*=====
18039 *          main
18040 *=====
18041 PUBLIC int main()
18042 {
18043 /* Main routine of the process manager. */
18044     int result, s, proc_nr;
18045     struct mproc *rmp;
18046     sigset_t sigset;
18047
18048     pm_init();           /* initialize process manager tables */
18049

```

```

18050 /* This is PM's main loop- get work and do it, forever and forever. */
18051 while (TRUE) {
18052     get_work();           /* wait for an PM system call */
18053
18054     /* Check for system notifications first. Special cases. */
18055     if (call_nr == SYN_ALARM) {
18056         pm_expire_timers(m_in.NOTIFY_TIMESTAMP);
18057         result = SUSPEND;           /* don't reply */
18058     } else if (call_nr == SYS_SIG) {        /* signals pending */
18059         sigset = m_in.NOTIFY_ARG;
18060         if (sigismember(&sigset, SIGKSIG)) (void) ksig_pending();
18061         result = SUSPEND;           /* don't reply */
18062     }
18063     /* Else, if the system call number is valid, perform the call. */
18064     else if ((unsigned) call_nr >= NCALLS) {
18065         result = ENOSYS;
18066     } else {
18067         result = (*call_vec[call_nr])();
18068     }
18069
18070     /* Send the results back to the user to indicate completion. */
18071     if (result != SUSPEND) setreply(who, result);
18072
18073     swap_in();           /* maybe a process can be swapped in? */
18074
18075     /* Send out all pending reply messages, including the answer to
18076      * the call just made above. The processes must not be swapped out.
18077      */
18078     for (proc_nr=0, rmp=mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
18079         /* In the meantime, the process may have been killed by a
18080          * signal (e.g. if a lethal pending signal was unblocked)
18081          * without the PM realizing it. If the slot is no longer in
18082          * use or just a zombie, don't try to reply.
18083         */
18084         if ((rmp->mp_flags & (REPLY | ONSWAP | IN_USE | ZOMBIE)) ==
18085             (REPLY | IN_USE)) {
18086             if ((s=send(proc_nr, &rmp->mp_reply)) != OK) {
18087                 panic(__FILE__, "PM can't reply to", proc_nr);
18088             }
18089             rmp->mp_flags &= ~REPLY;
18090         }
18091     }
18092 }
18093 return(OK);
18094 }

18095 /*=====
18096      *                         get_work
18097      *=====
18098 */
18099 PRIVATE void get_work()
18100 {
18101     /* Wait for the next message and extract useful information from it. */
18102     if (receive(ANY, &m_in) != OK) panic(__FILE__, "PM receive error", NO_NUM);
18103     who = m_in.m_source;           /* who sent the message */
18104     call_nr = m_in.m_type;        /* system call number */
18105
18106     /* Process slot of caller. Misuse PM's own process slot if the kernel is
18107      * calling. This can happen in case of synchronous alarms (CLOCK) or or
18108      * event like pending kernel signals (SYSTEM).
18109      */
18110     mp = &mproc[who < 0 ? PM_PROC_NR : who];
18111 }

18112 /*=====
18113      *                         setreply
18114      *=====
18115 */
18116 PUBLIC void setreply(proc_nr, result)

```

```

18117 int proc_nr;           /* process to reply to */
18118 int result;           /* result of call (usually OK or error #) */
18119 {
18120 /* Fill in a reply message to be sent later to a user process. System calls
18121 * may occasionally fill in other fields, this is only for the main return
18122 * value, and for setting the "must send reply" flag.
18123 */
18124 register struct mproc *rmp = &mproc[proc_nr];
18125
18126 rmp->mp_reply.reply_res = result;
18127 rmp->mp_flags |= REPLY;      /* reply pending */
18128
18129 if (rmp->mp_flags & ONSWAP)
18130     swap_inqueue(rmp);      /* must swap this process back in */
18131 }

18133 /*=====
18134 *          pm_init
18135 *=====
18136 PRIVATE void pm_init()
18137 {
18138 /* Initialize the process manager.
18139 * Memory use info is collected from the boot monitor, the kernel, and
18140 * all processes compiled into the system image. Initially this information
18141 * is put into an array mem_chunks. Elements of mem_chunks are struct memory,
18142 * and hold base, size pairs in units of clicks. This array is small, there
18143 * should be no more than 8 chunks. After the array of chunks has been built
18144 * the contents are used to initialize the hole list. Space for the hole list
18145 * is reserved as an array with twice as many elements as the maximum number
18146 * of processes allowed. It is managed as a linked list, and elements of the
18147 * array are struct hole, which, in addition to storage for a base and size in
18148 * click units also contain space for a link, a pointer to another element.
18149 */
18150 int s;
18151 static struct boot_image image[NR_BOOT_PROCS];
18152 register struct boot_image *ip;
18153 static char core_sigs[] = { SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
18154                           SIGEMT, SIGFPE, SIGUSR1, SIGSEGV, SIGUSR2 };
18155 static char ign_sigs[] = { SIGCHLD };
18156 register struct mproc *rmp;
18157 register char *sig_ptr;
18158 phys_clicks total_clicks, minix_clicks, free_clicks;
18159 message mess;
18160 struct mem_map mem_map[NR_LOCAL_SEGS];
18161 struct memory mem_chunks[NR_MEMS];
18162
18163 /* Initialize process table, including timers. */
18164 for (rmp=&mproc[0]; rmp<&mproc[NR_PROCS]; rmp++) {
18165     tmr_inittimer(&rmp->mp_timer);
18166 }
18167
18168 /* Build the set of signals which cause core dumps, and the set of signals
18169 * that are by default ignored.
18170 */
18171 sigemptyset(&core_sset);
18172 for (sig_ptr = core_sigs; sig_ptr < core_sigs+sizeof(core_sigs); sig_ptr++)
18173     sigaddset(&core_sset, *sig_ptr);
18174 sigemptyset(&ign_sset);
18175 for (sig_ptr = ign_sigs; sig_ptr < ign_sigs+sizeof(ign_sigs); sig_ptr++)
18176     sigaddset(&ign_sset, *sig_ptr);
18177
18178 /* Obtain a copy of the boot monitor parameters and the kernel info struct.
18179 * Parse the list of free memory chunks. This list is what the boot monitor
18180 * reported, but it must be corrected for the kernel and system processes.
18181 */
18182 if ((s=sys_getmonparams(monitor_params, sizeof(monitor_params))) != OK)
18183     panic(FILE, "get monitor params failed", s);

```

```

18184     get_mem_chunks(mem_chunks);
18185     if ((s=sys_getkinfo(&kinfo)) != OK)
18186         panic(FILE_, "get kernel info failed", s);
18187
18188     /* Get the memory map of the kernel to see how much memory it uses. */
18189     if ((s=get_mem_map(SYSTASK, mem_map)) != OK)
18190         panic(FILE_, "couldn't get memory map of SYSTASK", s);
18191     minix_ticks = (mem_map[S].mem_phys+mem_map[S].mem_len)-mem_map[T].mem_phys;
18192     patch_mem_chunks(mem_chunks, mem_map);
18193
18194     /* Initialize PM's process table. Request a copy of the system image table
18195      * that is defined at the kernel level to see which slots to fill in.
18196      */
18197     if (OK != (s=sys_getimage(image)))
18198         panic(FILE_, "couldn't get image table: %d\n", s);
18199     procs_in_use = 0;                                /* start populating table */
18200     printf("Building process table:");               /* show what's happening */
18201     for (ip = &image[0]; ip < &image[NR_BOOT_PROCS]; ip++) {
18202         if (ip->proc_nr >= 0) {                      /* task have negative nrs */
18203             procs_in_use += 1;                          /* found user process */
18204
18205             /* Set process details found in the image table. */
18206             rmp = &mproc[ip->proc_nr];
18207             strncpy(rmp->mp_name, ip->proc_name, PROC_NAME_LEN);
18208             rmp->mp_parent = RS_PROC_NR;
18209             rmp->mp_nice = get_nice_value(ip->priority);
18210             if (ip->proc_nr == INIT_PROC_NR) {           /* user process */
18211                 rmp->mp_pid = INIT_PID;
18212                 rmp->mp_flags |= IN_USE;
18213                 sigemptyset(&rmp->mp_ignore);
18214             }
18215             else {                                       /* system process */
18216                 rmp->mp_pid = get_free_pid();
18217                 rmp->mp_flags |= IN_USE | DONT_SWAP | PRIV_PROC;
18218                 sigfillset(&rmp->mp_ignore);
18219             }
18220             sigemptyset(&rmp->mp_sigmask);
18221             sigemptyset(&rmp->mp_catch);
18222             sigemptyset(&rmp->mp_sig2mess);
18223
18224             /* Get memory map for this process from the kernel. */
18225             if ((s=get_mem_map(ip->proc_nr, rmp->mp_seg)) != OK)
18226                 panic(FILE_, "couldn't get process entry", s);
18227             if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
18228             minix_ticks += rmp->mp_seg[S].mem_phys +
18229                         rmp->mp_seg[S].mem_len - rmp->mp_seg[T].mem_phys;
18230             patch_mem_chunks(mem_chunks, rmp->mp_seg);
18231
18232             /* Tell FS about this system process. */
18233             mess.PR_PROC_NR = ip->proc_nr;
18234             mess.PR_PID = rmp->mp_pid;
18235             if (OK != (s=send(FS_PROC_NR, &mess)))
18236                 panic(FILE_, "can't sync up with FS", s);
18237             printf("%s", ip->proc_name); /* display process name */
18238         }
18239     }
18240     printf(".\n");                                     /* last process done */
18241
18242     /* Override some details. PM is somewhat special. */
18243     mproc[PM_PROC_NR].mp_pid = PM_PID;                /* magically override pid */
18244     mproc[PM_PROC_NR].mp_parent = PM_PROC_NR;          /* PM doesn't have parent */
18245
18246     /* Tell FS that no more system processes follow and synchronize. */
18247     mess.PR_PROC_NR = NONE;
18248     if (sendrec(FS_PROC_NR, &mess) != OK || mess.m_type != OK)
18249         panic(FILE_, "can't sync up with FS", NO_NUM);
18250

```

```

18251 /* Initialize tables to all physical memory and print memory information. */
18252 printf("Physical memory:");
18253 mem_init(mem_chunks, &free_clicks);
18254 total_clicks = minix_clicks + free_clicks;
18255 printf(" total %u KB,", click_to_round_k(total_clicks));
18256 printf(" system %u KB,", click_to_round_k(minix_clicks));
18257 printf(" free %u KB.\n", click_to_round_k(free_clicks));
18258 }

18260 /*=====
18261 *           get_nice_value
18262 *=====
18263 PRIVATE int get_nice_value(queue)
18264 int queue;                                /* store mem chunks here */
18265 {
18266 /* Processes in the boot image have a priority assigned. The PM doesn't know
18267 * about priorities, but uses 'nice' values instead. The priority is between
18268 * MIN_USER_Q and MAX_USER_Q. We have to scale between PRIO_MIN and PRIO_MAX.
18269 */
18270     int nice_val = (queue - USER_Q) * (PRIO_MAX-PRIO_MIN+1) /
18271         (MIN_USER_Q-MAX_USER_Q+1);
18272     if (nice_val > PRIO_MAX) nice_val = PRIO_MAX; /* shouldn't happen */
18273     if (nice_val < PRIO_MIN) nice_val = PRIO_MIN; /* shouldn't happen */
18274     return nice_val;
18275 }

18277 /*=====
18278 *           get_mem_chunks
18279 *=====
18280 PRIVATE void get_mem_chunks(mem_chunks)
18281 struct memory *mem_chunks;                  /* store mem chunks here */
18282 {
18283 /* Initialize the free memory list from the 'memory' boot variable. Translate
18284 * the byte offsets and sizes in this list to clicks, properly truncated. Also
18285 * make sure that we don't exceed the maximum address space of the 286 or the
18286 * 8086, i.e. when running in 16-bit protected mode or real mode.
18287 */
18288     long base, size, limit;
18289     char *s, *end;                            /* use to parse boot variable */
18290     int i, done = 0;
18291     struct memory *memp;
18292
18293 /* Initialize everything to zero. */
18294 for (i = 0; i < NR_MEMS; i++) {
18295     memp = &mem_chunks[i];                /* next mem chunk is stored here */
18296     memp->base = memp->size = 0;
18297 }
18298
18299 /* The available memory is determined by MINIX' boot loader as a list of
18300 * (base:size)-pairs in bothead.s. The 'memory' boot variable is set in
18301 * in boot.s. The format is "b0:s0,b1:s1,b2:s2", where b0:s0 is low mem,
18302 * b1:s1 is mem between 1M and 16M, b2:s2 is mem above 16M. Pairs b1:s1
18303 * and b2:s2 are combined if the memory is adjacent.
18304 */
18305 s = find_param("memory");                  /* get memory boot variable */
18306 for (i = 0; i < NR_MEMS && !done; i++) {
18307     memp = &mem_chunks[i];                /* next mem chunk is stored here */
18308     base = size = 0;                      /* initialize next base:size pair */
18309     if (*s != 0) {                        /* get fresh data, unless at end */
18310
18311         /* Read fresh base and expect colon as next char. */
18312         base = strtoul(s, &end, 0x10);      /* get number */
18313         if (end != s && *end == ':') s = ++end; /* skip ':' */
18314         else *s=0;                         /* terminate, should not happen */
18315
18316         /* Read fresh size and expect comma or assume end. */
18317         size = strtoul(s, &end, 0x10);      /* get number */

```

```

18318         if (end != s && *end == ',') s = ++end;      /* skip ',' */
18319         else done = 1;
18320     }
18321     limit = base + size;
18322     base = (base + CLICK_SIZE-1) & ~(long)(CLICK_SIZE-1);
18323     limit &= ~(long)(CLICK_SIZE-1);
18324     if (limit <= base) continue;
18325     memp->base = base >> CLICK_SHIFT;
18326     memp->size = (limit - base) >> CLICK_SHIFT;
18327 }
18328 }

18330 /*=====
18331 *          patch_mem_chunks
18332 *=====
18333 PRIVATE void patch_mem_chunks(mem_chunks, map_ptr)
18334 struct memory *mem_chunks;           /* store mem chunks here */
18335 struct mem_map *map_ptr;            /* memory to remove */
18336 {
18337 /* Remove server memory from the free memory list. The boot monitor
18338 * promises to put processes at the start of memory chunks. The
18339 * tasks all use same base address, so only the first task changes
18340 * the memory lists. The servers and init have their own memory
18341 * spaces and their memory will be removed from the list.
18342 */
18343 struct memory *memp;
18344 for (memp = mem_chunks; memp < &mem_chunks[NR_MEMS]; memp++) {
18345     if (memp->base == map_ptr[T].mem_phys) {
18346         memp->base += map_ptr[T].mem_len + map_ptr[D].mem_len;
18347         memp->size -= map_ptr[T].mem_len + map_ptr[D].mem_len;
18348     }
18349 }
18350 }

```

+-----+  
servers/pm/forkexit.c  
+-----+

```

18400 /* This file deals with creating processes (via FORK) and deleting them (via
18401 * EXIT/WAIT). When a process forks, a new slot in the 'mproc' table is
18402 * allocated for it, and a copy of the parent's core image is made for the
18403 * child. Then the kernel and file system are informed. A process is removed
18404 * from the 'mproc' table when two events have occurred: (1) it has exited or
18405 * been killed by a signal, and (2) the parent has done a WAIT. If the process
18406 * exits first, it continues to occupy a slot until the parent does a WAIT.
18407 */
18408 /* The entry points into this file are:
18409 *   do_fork:    perform the FORK system call
18410 *   do_pm_exit: perform the EXIT system call (by calling pm_exit())
18411 *   pm_exit:    actually do the exiting
18412 *   do_wait:    perform the WAITPID or WAIT system call
18413 */
18414
18415 #include "pm.h"
18416 #include <sys/wait.h>
18417 #include <minix/callnr.h>
18418 #include <minix/com.h>
18419 #include <signal.h>
18420 #include "mproc.h"
18421 #include "param.h"
18422
18423 #define LAST_FEW          2 /* last few slots reserved for superuser */
18424
18425 FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child));
18426

```

```

18427 /*=====
18428 *          do_fork
18429 *=====
18430 PUBLIC int do_fork()
18431 {
18432 /* The process pointed to by 'mp' has forked. Create a child process. */
18433 register struct mproc *rmp; /* pointer to parent */
18434 register struct mproc *rmc; /* pointer to child */
18435 int child_nr, s;
18436 phys_clicks prog_clicks, child_base;
18437 phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
18438 pid_t new_pid;
18439
18440 /* If tables might fill up during FORK, don't even start since recovery half
18441 * way through is such a nuisance.
18442 */
18443 rmp = mp;
18444 if ((procs_in_use == NR_PROCS) ||
18445     (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0))
18446 {
18447     printf("PM: warning, process table is full!\n");
18448     return(EAGAIN);
18449 }
18450
18451 /* Determine how much memory to allocate. Only the data and stack need to
18452 * be copied, because the text segment is either shared or of zero length.
18453 */
18454 prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
18455 prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
18456 prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
18457 if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);
18458
18459 /* Create a copy of the parent's core image for the child. */
18460 child_abs = (phys_bytes) child_base << CLICK_SHIFT;
18461 parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
18462 s = sys_abscopy(parent_abs, child_abs, prog_bytes);
18463 if (s < 0) panic(FILE, "do_fork can't copy", s);
18464
18465 /* Find a slot in 'mproc' for the child process. A slot must exist. */
18466 for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
18467     if ( (rmc->mp_flags & IN_USE) == 0) break;
18468
18469 /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
18470 child_nr = (int)(rmc - mproc); /* slot number of the child */
18471 procs_in_use++;
18472 *rmc = *rmp; /* copy parent's process slot to child's */
18473 rmc->mp_parent = who; /* record child's parent */
18474 /* inherit only these flags */
18475 rmc->mp_flags &= (IN_USE|SEPARATE|PRIV_PROC|DONT_SWAP);
18476 rmc->mp_child_utime = 0; /* reset administration */
18477 rmc->mp_child_stime = 0; /* reset administration */
18478
18479 /* A separate I&D child keeps the parents text segment. The data and stack
18480 * segments must refer to the new copy.
18481 */
18482 if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
18483 rmc->mp_seg[D].mem_phys = child_base;
18484 rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
18485     (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
18486 rmc->mp_exitstatus = 0;
18487 rmc->mp_sigstatus = 0;
18488
18489 /* Find a free pid for the child and put it in the table. */
18490 new_pid = get_free_pid();
18491 rmc->mp_pid = new_pid; /* assign pid to child */
18492
18493 /* Tell kernel and file system about the (now successful) FORK. */

```

```

18494     sys_fork(who, child_nr);
18495     tell_fs(FORK, who, child_nr, rmc->mp_pid);
18496
18497     /* Report child's memory map to kernel. */
18498     sys_newmap(child_nr, rmc->mp_seg);
18499
18500     /* Reply to child to wake it up. */
18501     setreply(child_nr, 0);           /* only parent gets details */
18502     rmp->mp_reply.procnr = child_nr;    /* child's process number */
18503     return(new_pid);                /* child's pid */
18504 }

18505 /*=====
18506 *          do_pm_exit
18507 *=====
18508 PUBLIC int do_pm_exit()
18509 {
18510     /* Perform the exit(status) system call. The real work is done by pm_exit(),
18511     * which is also called when a process is killed by a signal.
18512     */
18513     pm_exit(rmp, m_in.status);
18514     return(SUSPEND);             /* can't communicate from beyond the grave */
18515 }
18516 */

18517 /*=====
18518 *          pm_exit
18519 *=====
18520 PUBLIC void pm_exit(rmp, exit_status)
18521 register struct mproc *rmp;      /* pointer to the process to be terminated */
18522 int exit_status;                /* the process' exit status (for parent) */
18523 {
18524     /* A process is done. Release most of the process' possessions. If its
18525     * parent is waiting, release the rest, else keep the process slot and
18526     * become a zombie.
18527     */
18528     register int proc_nr;
18529     int parent_waiting, right_child;
18530     pid_t pidarg, procgrp;
18531     struct mproc *p_mp;
18532     clock_t t[5];
18533
18534     proc_nr = (int) (rmp - mproc);        /* get process slot number */
18535
18536     /* Remember a session leader's process group. */
18537     procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
18538
18539     /* If the exited process has a timer pending, kill it. */
18540     if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);
18541
18542     /* Do accounting: fetch usage times and accumulate at parent. */
18543     sys_times(proc_nr, t);
18544     p_mp = &mproc[rmp->mp_parent];           /* process' parent */
18545     p_mp->mp_child_utime += t[0] + rmp->mp_child_utime; /* add user time */
18546     p_mp->mp_child_stime += t[1] + rmp->mp_child_stime; /* add system time */
18547
18548     /* Tell the kernel and FS that the process is no longer runnable. */
18549     tell_fs(EXIT, proc_nr, 0, 0); /* file system can free the proc slot */
18550     sys_exit(proc_nr);
18551
18552     /* Pending reply messages for the dead process cannot be delivered. */
18553     rmp->mp_flags &= ~REPLY;
18554
18555     /* Release the memory occupied by the child. */
18556     if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
18557         /* No other process shares the text segment, so free it. */
18558         free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
18559     }
18560 }
```

```

18561 /* Free the data and stack segments. */
18562 free_mem(rmp->mp_seg[D].mem_phys,
18563     rmp->mp_seg[S].mem_vir
18564     + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
18565
18566 /* The process slot can only be freed if the parent has done a WAIT. */
18567 rmp->mp_exitstatus = (char) exit_status;
18568
18569 pidarg = p_mp->mp_wpid;           /* who's being waited for? */
18570 parent_waiting = p_mp->mp_flags & WAITING;
18571 right_child =                  /* child meets one of the 3 tests? */
18572     (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_progrp);
18573
18574 if (parent_waiting & right_child) {
18575     cleanup(rmp);             /* tell parent and release child slot */
18576 } else {
18577     rmp->mp_flags = IN_USE|ZOMBIE; /* parent not waiting, zombify child */
18578     sig_proc(p_mp, SIGCHLD);    /* send parent a "child died" signal */
18579 }
18580
18581 /* If the process has children, disinherit them. INIT is the new parent. */
18582 for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
18583     if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
18584         /* 'rmp' now points to a child to be disinherited. */
18585         rmp->mp_parent = INIT_PROC_NR;
18586         parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
18587         if (parent_waiting && (rmp->mp_flags & ZOMBIE)) cleanup(rmp);
18588     }
18589 }
18590
18591 /* Send a hangup to the process' process group if it was a session leader. */
18592 if (procgrp != 0) check_sig(-procgrp, SIGHUP);
18593 }

18595 /*=====
18596 *          do_waitpid
18597 *=====
18598 PUBLIC int do_waitpid()
18599 {
18600 /* A process wants to wait for a child to terminate. If a child is already
18601 * waiting, go clean it up and let this WAIT call terminate. Otherwise,
18602 * really wait.
18603 * A process calling WAIT never gets a reply in the usual way at the end
18604 * of the main loop (unless WNOHANG is set or no qualifying child exists).
18605 * If a child has already exited, the routine cleanup() sends the reply
18606 * to awaken the caller.
18607 * Both WAIT and WAITPID are handled by this code.
18608 */
18609 register struct mproc *rp;
18610 int pidarg, options, children;
18611
18612 /* Set internal variables, depending on whether this is WAIT or WAITPID. */
18613 pidarg = (call_nr == WAIT ? -1 : m_in.pid); /* 1st param of waitpid */
18614 options = (call_nr == WAIT ? 0 : m_in.sig_nr); /* 3rd param of waitpid */
18615 if (pidarg == 0) pidarg = -mp->mp_progrp; /* pidarg < 0 ==> proc grp */
18616
18617 /* Is there a child waiting to be collected? At this point, pidarg != 0:
18618 *   * pidarg > 0 means pidarg is pid of a specific process to wait for
18619 *   * pidarg == -1 means wait for any child
18620 *   * pidarg < -1 means wait for any child whose process group = -pidarg
18621 */
18622 children = 0;
18623 for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
18624     if ((rp->mp_flags & IN_USE) && rp->mp_parent == who) {
18625         /* The value of pidarg determines which children qualify. */
18626         if (pidarg > 0 && pidarg != rp->mp_pid) continue;
18627         if (pidarg < -1 && -pidarg != rp->mp_progrp) continue;

```

```

18628             children++;           /* this child is acceptable */
18629             if (rp->mp_flags & ZOMBIE) {
18630                 /* This child meets the pid test and has exited. */
18631                 cleanup(rp);    /* this child has already exited */
18632                 return(SUSPEND);
18633             }
18634             if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
18635                 /* This child meets the pid test and is being traced.*/
18636                 mp->mp_reply.reply_res2 = 0177|(rp->mp_sigstatus << 8);
18637                 rp->mp_sigstatus = 0;
18638                 return(rp->mp_pid);
18639             }
18640         }
18641     }
18642 }
18643
18644 /* No qualifying child has exited. Wait for one, unless none exists. */
18645 if (children > 0) {
18646     /* At least 1 child meets the pid test exists, but has not exited.*/
18647     if (options & WNOHANG) return(0);    /* parent does not want to wait */
18648     mp->mp_flags |= WAITING;          /* parent wants to wait */
18649     mp->mp_wpid = (pid_t) pidarg;    /* save pid for later */
18650     return(SUSPEND);                /* do not reply, let it wait */
18651 } else {
18652     /* No child even meets the pid test. Return error immediately. */
18653     return(ECHILD);                  /* no - parent has no children */
18654 }
18655 }

18656 /*=====
18657 *                         cleanup
18658 *=====
18659 PRIVATE void cleanup(child)
18660 register struct mproc *child; /* tells which process is exiting */
18661 {
18662 /* Finish off the exit of a process. The process has exited or been killed
18663 * by a signal, and its parent is waiting.
18664 */
18665 struct mproc *parent = &mproc[child->mp_parent];
18666 int exitstatus;
18667
18668 /* Wake up the parent by sending the reply message. */
18669 exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
18670 parent->mp_reply.reply_res2 = exitstatus;
18671 setreply(child->mp_parent, child->mp_pid);
18672 parent->mp_flags &= ~WAITING; /* parent no longer waiting */
18673
18674 /* Release the process table entry and reinitialize some field. */
18675 child->mp_pid = 0;
18676 child->mp_flags = 0;
18677 child->mp_child_utime = 0;
18678 child->mp_child_stime = 0;
18679 procs_in_use--;
18680 }
18681 }
```

```
+++++
servers/pm/exec.c
+++++
```

```

18700 /* This file handles the EXEC system call. It performs the work as follows:
18701 *   - see if the permissions allow the file to be executed
18702 *   - read the header and extract the sizes
18703 *   - fetch the initial args and environment from the user space
18704 *   - allocate the memory for the new process
```

```

18705 *      - copy the initial stack from PM to the process
18706 *      - read in the text and data segments and copy to the process
18707 *      - take care of setuid and setgid bits
18708 *      - fix up 'mproc' table
18709 *      - tell kernel about EXEC
18710 *      - save offset to initial argc (for ps)
18711 *
18712 * The entry points into this file are:
18713 *      do_exec:    perform the EXEC system call
18714 *      rw_seg:    read or write a segment from or to a file
18715 *      find_share: find a process whose text segment can be shared
18716 */
18717
18718 #include "pm.h"
18719 #include <sys/stat.h>
18720 #include <minix/callnr.h>
18721 #include <minix/com.h>
18722 #include <a.out.h>
18723 #include <signal.h>
18724 #include <string.h>
18725 #include "mproc.h"
18726 #include "param.h"
18727
18728 FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
18729                     vir_bytes data_bytes, vir_bytes bss_bytes,
18730                     vir_bytes stk_bytes, phys_bytes tot_bytes) );
18731 FORWARD _PROTOTYPE( void patch_ptr, (char stack[ARG_MAX], vir_bytes base) );
18732 FORWARD _PROTOTYPE( int insert_arg, (char stack[ARG_MAX],
18733                     vir_bytes *stk_bytes, char *arg, int replace) );
18734 FORWARD _PROTOTYPE( char *patch_stack, (int fd, char stack[ARG_MAX],
18735                     vir_bytes *stk_bytes, char *script) );
18736 FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
18737                     vir_bytes *data_bytes, vir_bytes *bss_bytes,
18738                     phys_bytes *tot_bytes, long *sym_bytes, vir_ticks sc,
18739                     vir_bytes *pc) );
18740
18741 #define ESCRIPT (-2000) /* Returned by read_header for a #! script. */
18742 #define PTRSIZE sizeof(char *) /* Size of pointers in argv[] and envp[]. */
18743
18744 /*=====
18745 *          do_exec
18746 *=====
18747 PUBLIC int do_exec()
18748 {
18749 /* Perform the execve(name, argv, envp) call. The user library builds a
18750 * complete stack image, including pointers, args, environ, etc. The stack
18751 * is copied to a buffer inside PM, and then to the new core image.
18752 */
18753     register struct mproc *rmp;
18754     struct mproc *sh_mp;
18755     int m, r, fd, ft, sn;
18756     static char mbuf[ARG_MAX]; /* buffer for stack and zeroes */
18757     static char name_buf[PATH_MAX]; /* the name of the file to exec */
18758     char *new_sp, *name, *basename;
18759     vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
18760     phys_bytes tot_bytes; /* total space for program, including gap */
18761     long sym_bytes;
18762     vir_ticks sc;
18763     struct stat s_buf[2], *s_p;
18764     vir_bytes pc;
18765
18766 /* Do some validity checks. */
18767     rmp = mp;
18768     stk_bytes = (vir_bytes) m_in.stack_bytes;
18769     if (stk_bytes > ARG_MAX) return(ENOMEM); /* stack too big */
18770     if (m_in.exec_len <= 0 || m_in.exec_len > PATH_MAX) return(EINVAL);
18771
18772 /* Get the exec file name and see if the file is executable. */

```

```

18773     src = (vir_bytes) m_in.exec_name;
18774     dst = (vir_bytes) name_buf;
18775     r = sys_datacopy(who, (vir_bytes) src,
18776                         PM_PROC_NR, (vir_bytes) dst, (phys_bytes) m_in.exec_len);
18777     if (r != OK) return(r); /* file name not in user data segment */
18778
18779     /* Fetch the stack from the user before destroying the old core image. */
18780     src = (vir_bytes) m_in.stack_ptr;
18781     dst = (vir_bytes) mbuf;
18782     r = sys_datacopy(who, (vir_bytes) src,
18783                         PM_PROC_NR, (vir_bytes) dst, (phys_bytes)stk_bytes);
18784     /* can't fetch stack (e.g. bad virtual addr) */
18785     if (r != OK) return(EACCES);
18786
18787     r = 0; /* r = 0 (first attempt), or 1 (interpreted script) */
18788     name = name_buf; /* name of file to exec. */
18789     do {
18790         s_p = &s_buf[r];
18791         tell_fs(CHDIR, who, FALSE, 0); /* switch to the user's FS environ */
18792         fd = allowed(name, s_p, X_BIT); /* is file executable? */
18793         if (fd < 0) return(fd); /* file was not executable */
18794
18795         /* Read the file header and extract the segment sizes. */
18796         sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
18797
18798         m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
18799                         &tot_bytes, &sym_bytes, sc, &pc);
18800         if (m != ESCRIPT || ++r > 1) break;
18801     } while ((name = patch_stack(fd, mbuf, &stk_bytes, name_buf)) != NULL);
18802
18803     if (m < 0) {
18804         close(fd); /* something wrong with header */
18805         return(stk_bytes > ARG_MAX ? ENOMEM : ENOEXEC);
18806     }
18807
18808     /* Can the process' text be shared with that of one already running? */
18809     sh_mp = find_share(rmp, s_p->st_ino, s_p->st_dev, s_p->st_ctime);
18810
18811     /* Allocate new memory and release old memory. Fix map and tell kernel. */
18812     r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
18813     if (r != OK) {
18814         close(fd); /* insufficient core or program too big */
18815         return(r);
18816     }
18817
18818     /* Save file identification to allow it to be shared. */
18819     rmp->mp_ino = s_p->st_ino;
18820     rmp->mp_dev = s_p->st_dev;
18821     rmp->mp_ctime = s_p->st_ctime;
18822
18823     /* Patch up stack and copy it from PM to new core image. */
18824     vsp = (vir_bytes) rmp->mp_seg[S].mem_vir << CLICK_SHIFT;
18825     vsp += (vir_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT;
18826     vsp -= stk_bytes;
18827     patch_ptr(mbuf, vsp);
18828     src = (vir_bytes) mbuf;
18829     r = sys_datacopy(PM_PROC_NR, (vir_bytes) src,
18830                         who, (vir_bytes) vsp, (phys_bytes)stk_bytes);
18831     if (r != OK) panic(FILE, "do_exec stack copy err on", who);
18832
18833     /* Read in text and data segments. */
18834     if (sh_mp != NULL) {
18835         lseek(fd, (off_t) text_bytes, SEEK_CUR); /* shared: skip text */
18836     } else {
18837         rw_seg(0, fd, who, T, text_bytes);
18838     }
18839     rw_seg(0, fd, who, D, data_bytes);
18840

```

```

18841     close(fd);           /* don't need exec file any more */
18842
18843     /* Take care of setuid/setgid bits. */
18844     if ((rmp->mp_flags & TRACED) == 0) { /* suppress if tracing */
18845         if (s_buf[0].st_mode & I_SET_UID_BIT) {
18846             rmp->mp_effuid = s_buf[0].st_uid;
18847             tell_fs(SETUID, who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
18848         }
18849         if (s_buf[0].st_mode & I_SET_GID_BIT) {
18850             rmp->mp_effgid = s_buf[0].st_gid;
18851             tell_fs(SETGID, who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
18852         }
18853     }
18854
18855     /* Save offset to initial argc (for ps) */
18856     rmp->mp_procargs = vsp;
18857
18858     /* Fix 'mproc' fields, tell kernel that exec is done, reset caught sigs. */
18859     for (sn = 1; sn <= _NSIG; sn++) {
18860         if (sigismember(&rmp->mp_catch, sn)) {
18861             sigdelset(&rmp->mp_catch, sn);
18862             rmp->mp_sigact[sn].sa_handler = SIG_DFL;
18863             sigemptyset(&rmp->mp_sigact[sn].sa_mask);
18864         }
18865     }
18866
18867     rmp->mp_flags &= ~SEPARATE; /* turn off SEPARATE bit */
18868     rmp->mp_flags |= ft;        /* turn it on for separate I & D files */
18869     new_sp = (char *) vsp;
18870
18871     tell_fs(EXEC, who, 0, 0); /* allow FS to handle FD_CLOEXEC files */
18872
18873     /* System will save command line for debugging, ps(1) output, etc. */
18874     basename = strrchr(name, '/');
18875     if (basename == NULL) basename = name; else basename++;
18876     strncpy(rmp->mp_name, basename, PROC_NAME_LEN-1);
18877     rmp->mp_name[PROC_NAME_LEN] = '\0';
18878     sys_exec(who, new_sp, basename, pc);
18879
18880     /* Cause a signal if this process is traced. */
18881     if (rmp->mp_flags & TRACED) check_sig(rmp->mp_pid, SIGTRAP);
18882
18883     return(SUSPEND); /* no reply, new program just runs */
18884 }

18885 /*=====
18886 *          read_header
18887 *=====
18888 PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
18889                           tot_bytes, sym_bytes, sc, pc)
18890     int fd;           /* file descriptor for reading exec file */
18891     int *ft;          /* place to return ft number */
18892     vir_bytes *text_bytes; /* place to return text size */
18893     vir_bytes *data_bytes; /* place to return initialized data size */
18894     vir_bytes *bss_bytes; /* place to return bss size */
18895     phys_bytes *tot_bytes; /* place to return total size */
18896     long *sym_bytes; /* place to return symbol table size */
18897     vir_ticks sc; /* stack size in ticks */
18898     vir_bytes *pc; /* program entry point (initial PC) */
18899
18900 {
18901     /* Read the header and extract the text, data, bss and total sizes from it. */
18902
18903     int m, ct;
18904     vir_ticks tc, dc, s_vir, dvir;
18905     phys_ticks totc;
18906     struct exec hdr; /* a.out header is read in here */
18907

```

```

18908 /* Read the header and check the magic number. The standard MINIX header
18909 * is defined in <a.out.h>. It consists of 8 chars followed by 6 longs.
18910 * Then come 4 more longs that are not used here.
18911 * Byte 0: magic number 0x01
18912 * Byte 1: magic number 0x03
18913 * Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
18914 * Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
18915 * Motorola = 0x0B, Sun SPARC = 0x17
18916 * Byte 4: Header length = 0x20
18917 * Bytes 5-7 are not used.
18918 *
18919 * Now come the 6 longs
18920 * Bytes 8-11: size of text segments in bytes
18921 * Bytes 12-15: size of initialized data segment in bytes
18922 * Bytes 16-19: size of bss in bytes
18923 * Bytes 20-23: program entry point
18924 * Bytes 24-27: total memory allocated to program (text, data + stack)
18925 * Bytes 28-31: size of symbol table in bytes
18926 * The longs are represented in a machine dependent order,
18927 * little-endian on the 8088, big-endian on the 68000.
18928 * The header is followed directly by the text and data segments, and the
18929 * symbol table (if any). The sizes are given in the header. Only the
18930 * text and data segments are copied into memory by exec. The header is
18931 * used here only. The symbol table is for the benefit of a debugger and
18932 * is ignored here.
18933 */
18934
18935 if ((m= read(fd, &hdr, A_MINHDR)) < 2) return(ENOEXEC);
18936
18937 /* Interpreted script? */
18938 if (((char *) &hdr)[0] == '#' && ((char *) &hdr)[1] == '!') return(ESCRIPT);
18939
18940 if (m != A_MINHDR) return(ENOEXEC);
18941
18942 /* Check magic number, cpu type, and flags. */
18943 if (BADMAG(hdr)) return(ENOEXEC);
18944 if (hdr.a_cpu != A_I80386) return(ENOEXEC);
18945 if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);
18946
18947 *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0); /* separate I & D or not */
18948
18949 /* Get text and data sizes. */
18950 *text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
18951 *data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
18952 *bss_bytes = (vir_bytes) hdr.a_bss; /* bss size in bytes */
18953 *tot_bytes = hdr.a_total; /* total bytes to allocate for prog */
18954 *sym_bytes = hdr.a_syms; /* symbol table size in bytes */
18955 if (*tot_bytes == 0) return(ENOEXEC);
18956
18957 if (*ft != SEPARATE) {
18958     /* If I & D space is not separated, it is all considered data. Text=0*/
18959     *data_bytes += *text_bytes;
18960     *text_bytes = 0;
18961 }
18962 *pc = hdr.a_entry; /* initial address to start execution */
18963
18964 /* Check to see if segment sizes are feasible. */
18965 tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
18966 dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
18967 totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
18968 if (dc >= totc) return(ENOEXEC); /* stack must be at least 1 click */
18969 dvir = (*ft == SEPARATE ? 0 : tc);
18970 s_vir = dvir + (totc - sc);
18971 m = (dvir + dc > s_vir) ? ENOMEM : OK;
18972 ct = hdr.a_hdrlen & BYTE; /* header length */
18973 if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); /* skip unused hdr */
18974 return(m);
18975 }

```

```

18977 /*=====
18978 *          new_mem
18979 *=====
18980 PRIVATE int new_mem(sh_mp, text_bytes, data_bytes,
18981           bss_bytes,stk_bytes,tot_bytes)
18982 struct mproc *sh_mp;           /* text can be shared with this process */
18983 vir_bytes text_bytes;         /* text segment size in bytes */
18984 vir_bytes data_bytes;        /* size of initialized data in bytes */
18985 vir_bytes bss_bytes;         /* size of bss in bytes */
18986 vir_bytes stk_bytes;         /* size of initial stack segment in bytes */
18987 phys_bytes tot_bytes;        /* total memory to allocate, including gap */
18988 {
18989 /* Allocate new memory and release the old memory. Change the map and report
18990 * the new map to the kernel. Zero the new core image's bss, gap and stack.
18991 */
18992
18993 register struct mproc *rmp = mp;
18994 vir_ticks text_ticks, data_ticks, gap_ticks, stack_ticks, tot_ticks;
18995 phys_ticks new_base;
18996 phys_bytes bytes, base, bss_offset;
18997 int s;
18998
18999 /* No need to allocate text if it can be shared. */
19000 if (sh_mp != NULL) text_bytes = 0;
19001
19002 /* Allow the old data to be swapped out to make room. (Which is really a
19003 * waste of time, because we are going to throw it away anyway.)
19004 */
19005 rmp->mp_flags |= WAITING;
19006
19007 /* Acquire the new memory. Each of the 4 parts: text, (data+bss), gap,
19008 * and stack occupies an integral number of ticks, starting at click
19009 * boundary. The data and bss parts are run together with no space.
19010 */
19011 text_ticks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
19012 data_ticks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
19013 stack_ticks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
19014 tot_ticks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
19015 gap_ticks = tot_ticks - data_ticks - stack_ticks;
19016 if ( (int) gap_ticks < 0) return(ENOMEM);
19017
19018 /* Try to allocate memory for the new process. */
19019 new_base = alloc_mem(text_ticks + tot_ticks);
19020 if (new_base == NO_MEM) return(ENOMEM);
19021
19022 /* We've got memory for the new core image. Release the old one. */
19023 rmp = mp;
19024
19025 if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
19026     /* No other process shares the text segment, so free it. */
19027     free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
19028 }
19029 /* Free the data and stack segments. */
19030 free_mem(rmp->mp_seg[D].mem_phys,
19031           rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
19032
19033 /* We have now passed the point of no return. The old core image has been
19034 * forever lost, memory for a new core image has been allocated. Set up
19035 * and report new map.
19036 */
19037 if (sh_mp != NULL) {
19038     /* Share the text segment. */
19039     rmp->mp_seg[T] = sh_mp->mp_seg[T];
19040 } else {
19041     rmp->mp_seg[T].mem_phys = new_base;
19042     rmp->mp_seg[T].mem_vir = 0;
19043     rmp->mp_seg[T].mem_len = text_ticks;

```

```

19044 }
19045 rmp->mp_seg[D].mem_phys = new_base + text_clicks;
19046 rmp->mp_seg[D].mem_vir = 0;
19047 rmp->mp_seg[D].mem_len = data_clicks;
19048 rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
19049 rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
19050 rmp->mp_seg[S].mem_len = stack_clicks;
19051
19052 sys_newmap(who, rmp->mp_seg); /* report new map to the kernel */
19053
19054 /* The old memory may have been swapped out, but the new memory is real. */
19055 rmp->mp_flags &= ~(WAITING|ONSWAP|SWAPIN);
19056
19057 /* Zero the bss, gap, and stack segment. */
19058 bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
19059 base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
19060 bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
19061 base += bss_offset;
19062 bytes -= bss_offset;
19063
19064 if ((s=sys_memset(0, base, bytes)) != OK) {
19065     panic(__FILE__, "new_mem can't zero", s);
19066 }
19067
19068 return(OK);
19069 }

19071 /*=====
19072 *           patch_ptr
19073 *=====
19074 PRIVATE void patch_ptr(stack, base)
19075 char stack[ARG_MAX];          /* pointer to stack image within PM */
19076 vir_bytes base;             /* virtual address of stack base inside user */
19077 {
19078 /* When doing an exec(name, argv, envp) call, the user builds up a stack
19079 * image with arg and env pointers relative to the start of the stack. Now
19080 * these pointers must be relocated, since the stack is not positioned at
19081 * address 0 in the user's address space.
19082 */
19083
19084     char **ap, flag;
19085     vir_bytes v;
19086
19087     flag = 0;                  /* counts number of 0-pointers seen */
19088     ap = (char **) stack;      /* points initially to 'nargs' */
19089     ap++;                     /* now points to argv[0] */
19090     while (flag < 2) {
19091         if (ap >= (char **) &stack[ARG_MAX]) return; /* too bad */
19092         if (*ap != NULL) {
19093             v = (vir_bytes) *ap;    /* v is relative pointer */
19094             v += base;            /* relocate it */
19095             *ap = (char *) v;     /* put it back */
19096         } else {
19097             flag++;
19098         }
19099         ap++;
19100     }
19101 }

19103 /*=====
19104 *           insert_arg
19105 *=====
19106 PRIVATE int insert_arg(stack, stk_bytes, arg, replace)
19107 char stack[ARG_MAX];          /* pointer to stack image within PM */
19108 vir_bytes *stk_bytes;         /* size of initial stack */
19109 char *arg;                  /* argument to prepend/replace as new argv[0] */
19110 int replace;

```

```

19111 {
19112 /* Patch the stack so that arg will become argv[0]. Be careful, the stack may
19113 * be filled with garbage, although it normally looks like this:
19114 *      nargs argv[0] ... argv[nargs-1] NULL envp[0] ... NULL
19115 * followed by the strings "pointed" to by the argv[i] and the envp[i]. The
19116 * pointers are really offsets from the start of stack.
19117 * Return true iff the operation succeeded.
19118 */
19119     int offset, a0, a1, old_bytes = *stk_bytes;
19120
19121 /* Prepending arg adds at least one string and a zero byte. */
19122 offset = strlen(arg) + 1;
19123
19124 a0 = (int) ((char **) stack)[1]; /* argv[0] */
19125 if (a0 < 4 * PTRSIZE || a0 >= old_bytes) return(FALSE);
19126
19127 a1 = a0; /* a1 will point to the strings to be moved */
19128 if (replace) {
19129     /* Move a1 to the end of argv[0][] (argv[1] if nargs > 1). */
19130     do {
19131         if (a1 == old_bytes) return(FALSE);
19132         --offset;
19133     } while (stack[a1++] != 0);
19134 } else {
19135     offset += PTRSIZE; /* new argv[0] needs new pointer in argv[] */
19136     a0 += PTRSIZE; /* location of new argv[0][]. */
19137 }
19138
19139 /* stack will grow by offset bytes (or shrink by -offset bytes) */
19140 if ((*stk_bytes += offset) > ARG_MAX) return(FALSE);
19141
19142 /* Reposition the strings by offset bytes */
19143 memmove(stack + a1 + offset, stack + a1, old_bytes - a1);
19144
19145 strcpy(stack + a0, arg); /* Put arg in the new space. */
19146
19147 if (!replace) {
19148     /* Make space for a new argv[0]. */
19149     memmove(stack + 2 * PTRSIZE, stack + 1 * PTRSIZE, a0 - 2 * PTRSIZE);
19150
19151     ((char **) stack)[0]++; /* nargs++; */
19152 }
19153 /* Now patch up argv[] and envp[] by offset. */
19154 patch_ptr(stack, (vir_bytes) offset);
19155 ((char **) stack)[1] = (char *) a0; /* set argv[0] correctly */
19156 return(TRUE);
19157 }

19159 /*=====
19160 *          patch_stack
19161 *=====
19162 PRIVATE char *patch_stack(fd, stack, stk_bytes, script)
19163 int fd; /* file descriptor to open script file */
19164 char stack[ARG_MAX]; /* pointer to stack image within PM */
19165 vir_bytes *stk_bytes; /* size of initial stack */
19166 char *script; /* name of script to interpret */
19167 {
19168 /* Patch the argument vector to include the path name of the script to be
19169 * interpreted, and all strings on the #! line. Returns the path name of
19170 * the interpreter.
19171 */
19172 char *sp, *interp = NULL;
19173 int n;
19174 enum { INSERT=FALSE, REPLACE=TRUE };
19175
19176 /* Make script[] the new argv[0]. */
19177 if (!insert_arg(stack, stk_bytes, script, REPLACE)) return(NULL);

```

```

19178
19179     if (!seek(fd, 2L, 0) == -1           /* just behind the #! */
19180         || (n= read(fd, script, PATH_MAX)) < 0      /* read line one */
19181         || (sp= memchr(script, '\n', n)) == NULL)    /* must be a proper line */
19182             return(NULL);
19183
19184     /* Move sp backwards through script[], prepending each string to stack. */
19185     for (;;) {
19186         /* skip spaces behind argument. */
19187         while (sp > script && (*--sp == ' ' || *sp == '\t')) {}
19188         if (sp == script) break;
19189
19190         sp[1] = 0;
19191         /* Move to the start of the argument. */
19192         while (sp > script && sp[-1] != ' ' && sp[-1] != '\t') --sp;
19193
19194         interp = sp;
19195         if (!insert_arg(stack, stk_bytes, sp, INSERT)) return(NULL);
19196     }
19197
19198     /* Round *stk_bytes up to the size of a pointer for alignment constraints. */
19199     *stk_bytes= ((*stk_bytes + PTRSIZE - 1) / PTRSIZE) * PTRSIZE;
19200
19201     close(fd);
19202     return(interp);
19203 }

19204 /*=====
19205 *          rw_seg
19206 *=====
19207 PUBLIC void rw_seg(rw, fd, proc, seg, seg_bytes0)
19208 int rw;                      /* 0 = read, 1 = write */
19209 int fd;                      /* file descriptor to read from / write to */
19210 int proc;                    /* process number */
19211 int seg;                     /* T, D, or S */
19212 phys_bytes seg_bytes;        /* how much is to be transferred? */
19213 {
19214 /* Transfer text or data from/to a file and copy to/from a process segment.
19215 * This procedure is a little bit tricky. The logical way to transfer a
19216 * segment would be block by block and copying each block to/from the user
19217 * space one at a time. This is too slow, so we do something dirty here,
19218 * namely send the user space and virtual address to the file system in the
19219 * upper 10 bits of the file descriptor, and pass it the user virtual address
19220 * instead of a PM address. The file system extracts these parameters when
19221 * gets a read or write call from the process manager, which is the only
19222 * process that is permitted to use this trick. The file system then copies
19223 * the whole segment directly to/from user space, bypassing PM completely.
19224 *
19225 * The byte count on read is usually smaller than the segment count, because
19226 * a segment is padded out to a click multiple, and the data segment is only
19227 * partially initialized.
19228 */
19229
19230     int new_fd, bytes, r;
19231     char *ubuf_ptr;
19232     struct mem_map *sp = &mproc[proc].mp_seg[seg];
19233     phys_bytes seg_bytes = seg_bytes0;
19234
19235     new_fd = (proc << 7) | (seg << 5) | fd;
19236     ubuf_ptr = (char *) ((vir_bytes) sp->mem_vir << CLICK_SHIFT);
19237
19238     while (seg_bytes != 0) {
19239 #define PM_CHUNK_SIZE 8192
19240         bytes = MIN(INT_MAX / PM_CHUNK_SIZE) * PM_CHUNK_SIZE, seg_bytes);
19241         if (rw == 0) {
19242             r = read(new_fd, ubuf_ptr, bytes);
19243         } else {

```

```

19245         r = write(new_fd, ubuf_ptr, bytes);
19246     }
19247     if (r != bytes) break;
19248     ubuf_ptr += bytes;
19249     seg_bytes -= bytes;
19250   }
19251 }

19253 /*=====
19254 *          find_share
19255 *=====
19256 PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
19257 struct mproc *mp_ign;           /* process that should not be looked at */
19258 ino_t ino;                  /* parameters that uniquely identify a file */
19259 dev_t dev;
19260 time_t ctime;
19261 {
19262 /* Look for a process that is the file <ino, dev, ctime> in execution. Don't
19263 * accidentally "find" mp_ign, because it is the process on whose behalf this
19264 * call is made.
19265 */
19266 struct mproc *sh_mp;
19267 for (sh_mp = &mproc[0]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {
19268     if (!(sh_mp->mp_flags & SEPARATE)) continue;
19269     if (sh_mp == mp_ign) continue;
19270     if (sh_mp->mp_ino != ino) continue;
19271     if (sh_mp->mp_dev != dev) continue;
19272     if (sh_mp->mp_ctime != ctime) continue;
19273     return sh_mp;
19274 }
19275 }
19276 return(NULL);
19277 }

=====+
servers/pm/break.c
=====+

```

```

19300 /* The MINIX model of memory allocation reserves a fixed amount of memory for
19301 * the combined text, data, and stack segments. The amount used for a child
19302 * process created by FORK is the same as the parent had. If the child does
19303 * an EXEC later, the new size is taken from the header of the file EXEC'ed.
19304 *
19305 * The layout in memory consists of the text segment, followed by the data
19306 * segment, followed by a gap (unused memory), followed by the stack segment.
19307 * The data segment grows upward and the stack grows downward, so each can
19308 * take memory from the gap. If they meet, the process must be killed. The
19309 * procedures in this file deal with the growth of the data and stack segments.
19310 *
19311 * The entry points into this file are:
19312 *   do_brk:      BRK/SBRK system calls to grow or shrink the data segment
19313 *   adjust:      see if a proposed segment adjustment is allowed
19314 *   size_ok:     see if the segment sizes are feasible
19315 */
19316
19317 #include "pm.h"
19318 #include <signal.h>
19319 #include "mproc.h"
19320 #include "param.h"
19321
19322 #define DATA_CHANGED      1    /* flag value when data segment size changed */
19323 #define STACK_CHANGED      2    /* flag value when stack size changed */
19324
19325 /*=====
19326 *          do_brk
19327 *=====

```

```

19328 PUBLIC int do_brk()
19329 {
19330 /* Perform the brk(addr) system call.
19331 */
19332 * The call is complicated by the fact that on some machines (e.g., 8088),
19333 * the stack pointer can grow beyond the base of the stack segment without
19334 * anybody noticing it.
19335 * The parameter, 'addr' is the new virtual address in D space.
19336 */
19337
19338     register struct mproc *rmp;
19339     int r;
19340     vir_bytes v, new_sp;
19341     vir_ticks new_ticks;
19342
19343     rmp = mp;
19344     v = (vir_bytes) m_in.addr;
19345     new_ticks = (vir_ticks) ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT;
19346     if (new_ticks < rmp->mp_seg[D].mem_vir) {
19347         rmp->mp_reply.reply_ptr = (char *) -1;
19348         return(ENOMEM);
19349     }
19350     new_ticks -= rmp->mp_seg[D].mem_vir;
19351     if ((r = get_stack_ptr(who, &new_sp)) != OK) /* ask kernel for sp value */
19352         panic(__FILE__, "couldn't get stack pointer", r);
19353     r = adjust(rmp, new_ticks, new_sp);
19354     rmp->mp_reply.reply_ptr = (r == OK ? m_in.addr : (char *) -1);
19355     return(r); /* return new address or -1 */
19356 }

19358 /*=====
19359 *           adjust
19360 *=====
19361 PUBLIC int adjust(rmp, data_ticks, sp)
19362 register struct mproc *rmp; /* whose memory is being adjusted? */
19363 vir_ticks data_ticks; /* how big is data segment to become? */
19364 vir_bytes sp; /* new value of sp */
19365 {
19366 /* See if data and stack segments can coexist, adjusting them if need be.
19367 * Memory is never allocated or freed. Instead it is added or removed from the
19368 * gap between data segment and stack segment. If the gap size becomes
19369 * negative, the adjustment of data or stack fails and ENOMEM is returned.
19370 */
19371
19372     register struct mem_map *mem_sp, *mem_dp;
19373     vir_ticks sp_click, gap_base, lower, old_ticks;
19374     int changed, r, ft;
19375     long base_of_stack, delta; /* longs avoid certain problems */
19376
19377     mem_dp = &rmp->mp_seg[D]; /* pointer to data segment map */
19378     mem_sp = &rmp->mp_seg[S]; /* pointer to stack segment map */
19379     changed = 0; /* set when either segment changed */
19380
19381     if (mem_sp->mem_len == 0) return(OK); /* don't bother init */
19382
19383 /* See if stack size has gone negative (i.e., sp too close to 0xFFFF...) */
19384     base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
19385     sp_click = sp >> CLICK_SHIFT; /* click containing sp */
19386     if (sp_click >= base_of_stack) return(ENOMEM); /* sp too high */
19387
19388 /* Compute size of gap between stack and data segments. */
19389     delta = (long) mem_sp->mem_vir - (long) sp_click;
19390     lower = (delta > 0 ? sp_click : mem_sp->mem_vir);
19391
19392 /* Add a safety margin for future stack growth. Impossible to do right. */
19393 #define SAFETY_BYTES (384 * sizeof(char))
19394 #define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
19395     gap_base = mem_dp->mem_vir + data_ticks + SAFETY_CLICKS;

```

```

19396     if (lower < gap_base) return(ENOMEM); /* data and stack collided */
19397
19398     /* Update data length (but not data origin) on behalf of brk() system call. */
19399     old_clicks = mem_dp->mem_len;
19400     if (data_clicks != mem_dp->mem_len) {
19401         mem_dp->mem_len = data_clicks;
19402         changed |= DATA_CHANGED;
19403     }
19404
19405     /* Update stack length and origin due to change in stack pointer. */
19406     if (delta > 0) {
19407         mem_sp->mem_vir -= delta;
19408         mem_sp->mem_phys -= delta;
19409         mem_sp->mem_len += delta;
19410         changed |= STACK_CHANGED;
19411     }
19412
19413     /* Do the new data and stack segment sizes fit in the address space? */
19414     ft = (rmp->mp_flags & SEPARATE);
19415     r = (rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len >
19416          rmp->mp_seg[S].mem_vir) ? ENOMEM : OK;
19417     if (r == OK) {
19418         if (changed) sys_newmap((int)(rmp - mproc), rmp->mp_seg);
19419         return(OK);
19420     }
19421
19422     /* New sizes don't fit or require too many page/segment registers. Restore.*/
19423     if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
19424     if (changed & STACK_CHANGED) {
19425         mem_sp->mem_vir += delta;
19426         mem_sp->mem_phys += delta;
19427         mem_sp->mem_len -= delta;
19428     }
19429     return(ENOMEM);
19430 }

```

```
+++++
servers/pm/signal.c
+++++
```

```

19500 /* This file handles signals, which are asynchronous events and are generally
19501 * a messy and unpleasant business. Signals can be generated by the KILL
19502 * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
19503 * In all cases control eventually passes to check_sig() to see which processes
19504 * can be signaled. The actual signaling is done by sig_proc().
19505 *
19506 * The entry points into this file are:
19507 *   do_sigaction: perform the SIGACTION system call
19508 *   do_sigpending: perform the SIGPENDING system call
19509 *   do_sigprocmask: perform the SIGPROCMASK system call
19510 *   do_sigreturn: perform the SIGRETURN system call
19511 *   do_sigsuspend: perform the SIGSUSPEND system call
19512 *   do_kill: perform the KILL system call
19513 *   do_alarm: perform the ALARM system call by calling set_alarm()
19514 *   set_alarm: tell the clock task to start or stop a timer
19515 *   do_pause: perform the PAUSE system call
19516 *   ksig_pending: the kernel notified about pending signals
19517 *   sig_proc: interrupt or terminate a signaled process
19518 *   check_sig: check which processes to signal with sig_proc()
19519 *   check_pending: check if a pending signal can now be delivered
19520 */
19521
19522 #include "pm.h"
19523 #include <sys/stat.h>
19524 #include <sys/ptrace.h>
19525 #include <minix/callnr.h>

```

```

19526 #include <minix/com.h>
19527 #include <signal.h>
19528 #include <sys/sigcontext.h>
19529 #include <string.h>
19530 #include "mproc.h"
19531 #include "param.h"
19532
19533 #define CORE_MODE      0777 /* mode to use on core image files */
19534 #define DUMPED          0200 /* bit set in status when core dumped */
19535
19536 FORWARD _PROTOTYPE( void dump_core, (struct mproc *rmp) );
19537 FORWARD _PROTOTYPE( void unpause, (int pro) );
19538 FORWARD _PROTOTYPE( void handle_sig, (int proc_nr, sigset_t sig_map) );
19539 FORWARD _PROTOTYPE( void cause_sigalarm, (struct timer *tp) );
19540
19541 /*=====
19542 *           do_sigaction
19543 *=====
19544 PUBLIC int do_sigaction()
19545 {
19546     int r;
19547     struct sigaction svec;
19548     struct sigaction *svp;
19549
19550     if (m_in.sig_nr == SIGKILL) return(OK);
19551     if (m_in.sig_nr < 1 || m_in.sig_nr > _NSIG) return (EINVAL);
19552     svp = &mp->mp_sigact[m_in.sig_nr];
19553     if ((struct sigaction *) m_in.sig_osa != (struct sigaction *) NULL) {
19554         r = sys_datacopy(PM_PROC_NR,(vir_bytes) svp,
19555             who, (vir_bytes) m_in.sig_osa, (phys_bytes) sizeof(svec));
19556         if (r != OK) return(r);
19557     }
19558
19559     if ((struct sigaction *) m_in.sig_nsa == (struct sigaction *) NULL)
19560         return(OK);
19561
19562     /* Read in the sigaction structure. */
19563     r = sys_datacopy(who, (vir_bytes) m_in.sig_nsa,
19564                     PM_PROC_NR, (vir_bytes) &svec, (phys_bytes) sizeof(svec));
19565     if (r != OK) return(r);
19566
19567     if (svec.sa_handler == SIG_IGN) {
19568         sigaddset(&mp->mp_ignore, m_in.sig_nr);
19569         sigdelset(&mp->mp_siginpending, m_in.sig_nr);
19570         sigdelset(&mp->mp_catch, m_in.sig_nr);
19571         sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
19572     } else if (svec.sa_handler == SIG_DFL) {
19573         sigdelset(&mp->mp_ignore, m_in.sig_nr);
19574         sigdelset(&mp->mp_catch, m_in.sig_nr);
19575         sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
19576     } else if (svec.sa_handler == SIG_MESS) {
19577         if (! (mp->mp_flags & PRIV_PROC)) return(EPERM);
19578         sigdelset(&mp->mp_ignore, m_in.sig_nr);
19579         sigaddset(&mp->mp_sig2mess, m_in.sig_nr);
19580         sigdelset(&mp->mp_catch, m_in.sig_nr);
19581     } else {
19582         sigdelset(&mp->mp_ignore, m_in.sig_nr);
19583         sigaddset(&mp->mp_catch, m_in.sig_nr);
19584         sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
19585     }
19586     mp->mp_sigact[m_in.sig_nr].sa_handler = svec.sa_handler;
19587     sigdelset(&svec.sa_mask, SIGKILL);
19588     mp->mp_sigact[m_in.sig_nr].sa_mask = svec.sa_mask;
19589     mp->mp_sigact[m_in.sig_nr].sa_flags = svec.sa_flags;
19590     mp->mp_sigreturn = (vir_bytes) m_in.sig_ret;
19591     return(OK);
19592 }
```

```

19594 /*-----*
19595 * do_sigpending *
19596 *-----*/
19597 PUBLIC int do_sigpending()
19598 {
19599     mp->mp_reply.reply_mask = (long) mp->mp_sigpending;
19600     return OK;
19601 }

19603 /*-----*
19604 * do_sigprocmask *
19605 *-----*/
19606 PUBLIC int do_sigprocmask()
19607 {
19608     /* Note that the library interface passes the actual mask in sigmask_set,
19609     * not a pointer to the mask, in order to save a copy. Similarly,
19610     * the old mask is placed in the return message which the library
19611     * interface copies (if requested) to the user specified address.
19612     */
19613     /* The library interface must set SIG_INQUIRE if the 'act' argument
19614     * is NULL.
19615     */
19616
19617     int i;
19618
19619     mp->mp_reply.reply_mask = (long) mp->mp_sigmask;
19620
19621     switch (m_in.sig_how) {
19622         case SIG_BLOCK:
19623             sigdelset((sigset_t *)&m_in.sig_set, SIGKILL);
19624             for (i = 1; i <= _NSIG; i++) {
19625                 if (sigismember((sigset_t *)&m_in.sig_set, i))
19626                     sigaddset(&mp->mp_sigmask, i);
19627             }
19628             break;
19629
19630         case SIG_UNBLOCK:
19631             for (i = 1; i <= _NSIG; i++) {
19632                 if (sigismember((sigset_t *)&m_in.sig_set, i))
19633                     sigdelset(&mp->mp_sigmask, i);
19634             }
19635             check_pending(mp);
19636             break;
19637
19638         case SIG_SETMASK:
19639             sigdelset((sigset_t *) &m_in.sig_set, SIGKILL);
19640             mp->mp_sigmask = (sigset_t) m_in.sig_set;
19641             check_pending(mp);
19642             break;
19643
19644         case SIG_INQUIRE:
19645             break;
19646
19647         default:
19648             return(EINVAL);
19649             break;
19650     }
19651     return OK;
19652 }

19654 /*-----*
19655 * do_sigsuspend *
19656 *-----*/
19657 PUBLIC int do_sigsuspend()
19658 {
19659     mp->mp_sigmask2 = mp->mp_sigmask; /* save the old mask */
19660     mp->mp_sigmask = (sigset_t) m_in.sig_set;

```

```

19661     sigdelset(&mp->mp_sigmask, SIGKILL);
19662     mp->mp_flags |= SIGSUSPENDED;
19663     check_pending(mp);
19664     return(SUSPEND);
19665 }

19666 /*=====
19667 *          do_sigreturn
19668 *=====
19669 */
19670 PUBLIC int do_sigreturn()
19671 {
19672 /* A user signal handler is done. Restore context and check for
19673 * pending unblocked signals.
19674 */
19675
19676     int r;
19677
19678     mp->mp_sigmask = (sigset_t) m_in.sig_set;
19679     sigdelset(&mp->mp_sigmask, SIGKILL);
19680
19681     r = sys_sigreturn(who, (struct sigmsg *) m_in.sig_context);
19682     check_pending(mp);
19683     return(r);
19684 }

19685 /*=====
19686 *          do_kill
19687 *=====
19688 */
19689 PUBLIC int do_kill()
19690 {
19691 /* Perform the kill(pid, signo) system call. */
19692
19693     return check_sig(m_in.pid, m_in.sig_nr);
19694 }

19695 /*=====
19696 *          ksig_pending
19697 *=====
19698 */
19699 PUBLIC int ksig_pending()
19700 {
19701 /* Certain signals, such as segmentation violations originate in the kernel.
19702 * When the kernel detects such signals, it notifies the PM to take further
19703 * action. The PM requests the kernel to send messages with the process
19704 * slot and bit map for all signaled processes. The File System, for example,
19705 * uses this mechanism to signal writing on broken pipes (SIGPIPE).
19706 *
19707 * The kernel has notified the PM about pending signals. Request pending
19708 * signals until all signals are handled. If there are no more signals,
19709 * NONE is returned in the process number field.
19710 */
19711     int proc_nr;
19712     sigset_t sig_map;
19713
19714     while (TRUE) {
19715         sys_getksig(&proc_nr, &sig_map); /* get an arbitrary pending signal */
19716         if (NONE == proc_nr) {           /* stop if no more pending signals */
19717             break;
19718         } else {
19719             handle_sig(proc_nr, sig_map); /* handle the received signal */
19720             sys_endksig(proc_nr);        /* tell kernel it's done */
19721         }
19722     }
19723     return(SUSPEND);               /* prevents sending reply */
19724 }

19725 /*=====
19726 *          handle_sig
19727 *=====
19728 */

```

```

19729 PRIVATE void handle_sig(proc_nr, sig_map)
19730     int proc_nr;
19731     sigset_t sig_map;
19732 {
19733     register struct mproc *rmp;
19734     int i;
19735     pid_t proc_id, id;
19736
19737     rmp = &mproc[proc_nr];
19738     if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return;
19739     proc_id = rmp->mp_pid;
19740     mp = &mproc[0];           /* pretend signals are from PM */
19741     mp->mp_procgrp = rmp->mp_procgrp; /* get process group right */
19742
19743     /* Check each bit in turn to see if a signal is to be sent. Unlike
19744      * kill(), the kernel may collect several unrelated signals for a
19745      * process and pass them to PM in one blow. Thus loop on the bit
19746      * map. For SIGINT and SIGQUIT, use proc_id 0 to indicate a broadcast
19747      * to the recipient's process group. For SIGKILL, use proc_id -1 to
19748      * indicate a systemwide broadcast.
19749      */
19750     for (i = 1; i <= _NSIG; i++) {
19751         if (!sigismember(&sig_map, i)) continue;
19752         switch (i) {
19753             case SIGINT:
19754             case SIGQUIT:
19755                 id = 0; break; /* broadcast to process group */
19756             case SIGKILL:
19757                 id = -1; break; /* broadcast to all except INIT */
19758             default:
19759                 id = proc_id;
19760                 break;
19761         }
19762         check_sig(id, i);
19763     }
19764 }

19765 /*=====
19766      *          do_alarm
19767      *=====
19768 */
19769 PUBLIC int do_alarm()
19770 {
19771     /* Perform the alarm(seconds) system call. */
19772     return(set_alarm(who, m_in.seconds));
19773 }

19774 /*=====
19775      *          set_alarm
19776      *=====
19777 */
19778 PUBLIC int set_alarm(proc_nr, sec)
19779     int proc_nr;           /* process that wants the alarm */
19780     int sec;                /* how many seconds delay before the signal */
19781 {
19782     /* This routine is used by do_alarm() to set the alarm timer. It is also used
19783      * to turn the timer off when a process exits with the timer still on.
19784      */
19785     clock_t ticks;        /* number of ticks for alarm */
19786     clock_t exptime;      /* needed for remaining time on previous alarm */
19787     clock_t uptime;       /* current system time */
19788     int remaining;        /* previous time left in seconds */
19789     int s;
19790
19791     /* First determine remaining time of previous alarm, if set. */
19792     if (mproc[proc_nr].mp_flags & ALARM_ON) {
19793         if ( (s=getuptime(&uptime)) != OK)
19794             panic(__FILE__,"set_alarm couldn't get uptime", s);

```

```

19795     exptime = *tmr_exp_time(&mproc[proc_nr].mp_timer);
19796     remaining = (int) ((exptime - uptime + (HZ-1))/HZ);
19797     if (remaining < 0) remaining = 0;
19798 } else {
19799     remaining = 0;
19800 }
19801
19802 /* Tell the clock task to provide a signal message when the time comes.
19803 */
19804 /* Large delays cause a lot of problems. First, the alarm system call
19805 takes an unsigned seconds count and the library has cast it to an int.
19806 That probably works, but on return the library will convert "negative"
19807 unsigneds to errors. Presumably no one checks for these errors, so
19808 force this call through. Second, If unsigned and long have the same
19809 size, converting from seconds to ticks can easily overflow. Finally,
19810 the kernel has similar overflow bugs adding ticks.
19811 */
19812 /* Fixing this requires a lot of ugly casts to fit the wrong interface
19813 types and to avoid overflow traps. ALRM_EXP_TIME has the right type
19814 (clock_t) although it is declared as long. How can variables like
19815 this be declared properly without combinatorial explosion of message
19816 types?
19817 */
19818 ticks = (clock_t) (HZ * (unsigned long) (unsigned) sec);
19819 if ( (unsigned long) ticks / HZ != (unsigned) sec)
19820     ticks = LONG_MAX;           /* eternity (really TMR_NEVER) */
19821
19822 if (ticks != 0) {
19823     pm_set_timer(&mproc[proc_nr].mp_timer, ticks, cause_sigalarm, proc_nr);
19824     mproc[proc_nr].mp_flags |= ALARM_ON;
19825 } else if (mproc[proc_nr].mp_flags & ALARM_ON) {
19826     pm_cancel_timer(&mproc[proc_nr].mp_timer);
19827     mproc[proc_nr].mp_flags &= ~ALARM_ON;
19828 }
19829 return(remaining);
19830 }

19832 /*=====
19833 *          cause_sigalarm
19834 *=====
19835 PRIVATE void cause_sigalarm(tp)
19836 struct timer *tp;
19837 {
19838     int proc_nr;
19839     register struct mproc *rmp;
19840
19841     proc_nr = tmr_arg(tp)->ta_int;      /* get process from timer */
19842     rmp = &mproc[proc_nr];
19843
19844     if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return;
19845     if ((rmp->mp_flags & ALARM_ON) == 0) return;
19846     rmp->mp_flags &= ~ALARM_ON;
19847     check_sig(rmp->mp_pid, SIGALRM);
19848 }

19850 /*=====
19851 *          do_pause
19852 *=====
19853 PUBLIC int do_pause()
19854 {
19855     /* Perform the pause() system call. */
19856
19857     mp->mp_flags |= PAUSED;
19858     return(SUSPEND);
19859 }
```

```

19861 /*=====
19862 *          sig_proc
19863 *=====
19864 PUBLIC void sig_proc(rmp, signo)
19865 register struct mproc *rmp;      /* pointer to the process to be signaled */
19866 int signo;                      /* signal to send to process (1 to _NSIG) */
19867 {
19868 /* Send a signal to a process. Check to see if the signal is to be caught,
19869 * ignored, tranformed into a message (for system processes) or blocked.
19870 * - If the signal is to be transformed into a message, request the KERNEL to
19871 * send the target process a system notification with the pending signal as an
19872 * argument.
19873 * - If the signal is to be caught, request the KERNEL to push a sigcontext
19874 * structure and a sigframe structure onto the catcher's stack. Also, KERNEL
19875 * will reset the program counter and stack pointer, so that when the process
19876 * next runs, it will be executing the signal handler. When the signal handler
19877 * returns, sigreturn(2) will be called. Then KERNEL will restore the signal
19878 * context from the sigcontext structure.
19879 * If there is insufficient stack space, kill the process.
19880 */
19881
19882     vir_bytes new_sp;
19883     int s;
19884     int slot;
19885     int sigflags;
19886     struct sigmsg sm;
19887
19888     slot = (int) (rmp - mproc);
19889     if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) {
19890         printf("PM: signal %d sent to %s process %d\n",
19891             signo, (rmp->mp_flags & ZOMBIE) ? "zombie" : "dead", slot);
19892         panic(__FILE__, "", NO_NUM);
19893     }
19894     if ((rmp->mp_flags & TRACED) && signo != SIGKILL) {
19895         /* A traced process has special handling. */
19896         unpause(slot);
19897         stop_proc(rmp, signo); /* a signal causes it to stop */
19898         return;
19899     }
19900     /* Some signals are ignored by default. */
19901     if (sigismember(&rmp->mp_ignore, signo)) {
19902         return;
19903     }
19904     if (sigismember(&rmp->mp_sigmask, signo)) {
19905         /* Signal should be blocked. */
19906         sigaddset(&rmp->mp_sigpending, signo);
19907         return;
19908     }
19909     sigflags = rmp->mp_sigact[signo].sa_flags;
19910     if (sigismember(&rmp->mp_catch, signo)) {
19911         if (rmp->mp_flags & SIGSUSPENDED)
19912             sm.sm_mask = rmp->mp_sigmask2;
19913         else
19914             sm.sm_mask = rmp->mp_sigmask;
19915         sm.sm_signo = signo;
19916         sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
19917         sm.sm_sigreturn = rmp->mp_sigreturn;
19918         if ((s=get_stack_ptr(slot, &new_sp)) != OK)
19919             panic(__FILE__, "couldn't get new stack pointer", s);
19920         sm.sm_stkptr = new_sp;
19921
19922         /* Make room for the sigcontext and sigframe struct. */
19923         new_sp -= sizeof(struct sigcontext)
19924                         + 3 * sizeof(char *) + 2 * sizeof(int);
19925
19926         if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
19927             goto determinate;

```

```

19928
19929     rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
19930     if (sigflags & SA_NODEFER)
19931         sigdelset(&rmp->mp_sigmask, signo);
19932     else
19933         sigaddset(&rmp->mp_sigmask, signo);
19934
19935     if (sigflags & SA_RESETHAND) {
19936         sigdelset(&rmp->mp_catch, signo);
19937         rmp->mp_sigact[signo].sa_handler = SIG_DFL;
19938     }
19939
19940     if (OK == (s=sys_sigsend(slot, &sm))) {
19941
19942         sigdelset(&rmp->mp_sigsending, signo);
19943         /* If process is hanging on PAUSE, WAIT, SIGSUSPEND, tty,
19944            * pipe, etc., release it.
19945            */
19946         unpause(slot);
19947         return;
19948     }
19949     panic(__FILE__, "warning, sys_sigsend failed", s);
19950 }
19951 else if (sigismember(&rmp->mp_sig2mess, signo)) {
19952     if (OK != (s=sys_kill(slot, signo)))
19953         panic(__FILE__, "warning, sys_kill failed", s);
19954     return;
19955 }
19956
19957 determinate:
19958     /* Signal should not or cannot be caught. Take default action. */
19959     if (sigismember(&ign_sset, signo)) return;
19960
19961     rmp->mp_sigstatus = (char) signo;
19962     if (sigismember(&core_sset, signo)) {
19963         /* Switch to the user's FS environment and dump core. */
19964         tell_fs(CHDIR, slot, FALSE, 0);
19965         dump_core(rmp);
19966     }
19967     pm_exit(rmp, 0);           /* terminate process */
19968 }

19970 /*=====
19971 *          check_sig
19972 *=====
19973 PUBLIC int check_sig(proc_id, signo)
19974 pid_t proc_id;           /* pid of proc to sig, or 0 or -1, or -pgrp */
19975 int signo;              /* signal to send to process (0 to _NSIG) */
19976 {
19977     /* Check to see if it is possible to send a signal. The signal may have to be
19978     * sent to a group of processes. This routine is invoked by the KILL system
19979     * call, and also when the kernel catches a DEL or other signal.
19980     */
19981     register struct mproc *rmp;
19982     int count;                  /* count # of signals sent */
19983     int error_code;
19984
19985     if (signo < 0 || signo > _NSIG) return(EINVAL);
19986
19987     /* Return EINVAL for attempts to send SIGKILL to INIT alone. */
19988     if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);
19989
19990     /* Search the proc table for processes to signal. (See forkexit.c about
19991     * pid magic.)
19992     */
19993     count = 0;

```

```

19995     error_code = ESRCH;
19996     for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
19997         if (!(rmp->mp_flags & IN_USE)) continue;
19998         if ((rmp->mp_flags & ZOMBIE) && signo != 0) continue;
19999
20000         /* Check for selection. */
20001         if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
20002         if (proc_id == 0 && mp->mp_progrp != rmp->mp_progrp) continue;
20003         if (proc_id == -1 && rmp->mp_pid <= INIT_PID) continue;
20004         if (proc_id < -1 && rmp->mp_progrp != -proc_id) continue;
20005
20006         /* Check for permission. */
20007         if (mp->mp_effuid != SUPER_USER
20008             && mp->mp_realuid != rmp->mp_realuid
20009             && mp->mp_effuid != rmp->mp_realuid
20010             && mp->mp_realuid != rmp->mp_effuid
20011             && mp->mp_effuid != rmp->mp_effuid) {
20012             error_code = EPERM;
20013             continue;
20014         }
20015
20016         count++;
20017         if (signo == 0) continue;
20018
20019         /* 'sig_proc' will handle the disposition of the signal. The
20020          * signal may be caught, blocked, ignored, or cause process
20021          * termination, possibly with core dump.
20022          */
20023         sig_proc(rmp, signo);
20024
20025         if (proc_id > 0) break; /* only one process being signaled */
20026     }
20027
20028     /* If the calling process has killed itself, don't reply. */
20029     if ((mp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return(SUSPEND);
20030     return(count > 0 ? OK : error_code);
20031 }
20032 /*=====
20033      *                         check_pending
20034      *=====
20035 */
20036 PUBLIC void check_pending(rmp)
20037 register struct mproc *rmp;
20038 {
20039     /* Check to see if any pending signals have been unblocked. The
20040      * first such signal found is delivered.
20041      *
20042      * If multiple pending unmasked signals are found, they will be
20043      * delivered sequentially.
20044      *
20045      * There are several places in this file where the signal mask is
20046      * changed. At each such place, check_pending() should be called to
20047      * check for newly unblocked signals.
20048      */
20049
20050     int i;
20051
20052     for (i = 1; i <= _NSIG; i++) {
20053         if (sigismember(&rmp->mp_sigpending, i) &&
20054             !sigismember(&rmp->mp_sigmask, i)) {
20055             sigdelset(&rmp->mp_sigpending, i);
20056             sig_proc(rmp, i);
20057             break;
20058         }
20059     }
20060 }

```

```

20062 /*=====
20063 *          unpause
20064 *=====
20065 PRIVATE void unpause(pro)
20066 int pro;           /* which process number */
20067 {
20068 /* A signal is to be sent to a process. If that process is hanging on a
20069 * system call, the system call must be terminated with EINTR. Possible
20070 * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
20071 * First check if the process is hanging on an PM call. If not, tell FS,
20072 * so it can check for READs and WRITEs from pipes, ttys and the like.
20073 */
20074
20075 register struct mproc *rmp;
20076
20077 rmp = &mproc[pro];
20078
20079 /* Check to see if process is hanging on a PAUSE, WAIT or SIGSUSPEND call. */
20080 if (rmp->mp_flags & (PAUSED | WAITING | SIGSUSPENDED)) {
20081     rmp->mp_flags &= ~(PAUSED | WAITING | SIGSUSPENDED);
20082     setreply(pro, EINTR);
20083     return;
20084 }
20085
20086 /* Process is not harging on an PM call. Ask FS to take a look. */
20087 tell_fs(UNPAUSE, pro, 0, 0);
20088 }

20090 /*=====
20091 *          dump_core
20092 *=====
20093 PRIVATE void dump_core(rmp)
20094 register struct mproc *rmp; /* whose core is to be dumped */
20095 {
20096 /* Make a core dump on the file "core", if possible. */
20097
20098     int s, fd, seg, slot;
20099     vir_bytes current_sp;
20100     long trace_data, trace_off;
20101
20102     slot = (int) (rmp - mproc);
20103
20104     /* Can core file be written? We are operating in the user's FS environment,
20105     * so no special permission checks are needed.
20106     */
20107     if (rmp->mp_realuid != rmp->mp_effuid) return;
20108     if ((fd = open(core_name, O_WRONLY | O_CREAT | O_TRUNC | O_NONBLOCK,
20109                           CORE_MODE)) < 0) return;
20110     rmp->mp_sigstatus |= DUMPED;
20111
20112     /* Make sure the stack segment is up to date.
20113     * We don't want adjust() to fail unless current_sp is preposterous,
20114     * but it might fail due to safety checking. Also, we don't really want
20115     * the adjust() for sending a signal to fail due to safety checking.
20116     * Maybe make SAFETY_BYTES a parameter.
20117     */
20118     if ((s=get_stack_ptr(slot, &current_sp)) != OK)
20119         panic(FILE, "couldn't get new stack pointer", s);
20120     adjust(rmp, rmp->mp_seg[D].mem.len, current_sp);
20121
20122     /* Write the memory map of all segments to begin the core file. */
20123     if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
20124         != (unsigned) sizeof rmp->mp_seg) {
20125         close(fd);
20126         return;
20127     }

```

```

20128
20129     /* Write out the whole kernel process table entry to get the regs. */
20130     trace_off = 0;
20131     while (sys_trace(T_GETUSER, slot, trace_off, &trace_data) == OK) {
20132         if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
20133             != (unsigned) sizeof (long)) {
20134             close(fd);
20135             return;
20136         }
20137         trace_off += sizeof (long);
20138     }
20139
20140     /* Loop through segments and write the segments themselves out. */
20141     for (seg = 0; seg < NR_LOCAL_SEGS; seg++) {
20142         rw_seg(1, fd, slot, seg,
20143             (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT);
20144     }
20145     close(fd);
20146 }
```

```
+++++
servers/pm/timers.c
+++++
```

```

20200 /* PM watchdog timer management. These functions in this file provide
20201 * a convenient interface to the timers library that manages a list of
20202 * watchdog timers. All details of scheduling an alarm at the CLOCK task
20203 * are hidden behind this interface.
20204 * Only system processes are allowed to set an alarm timer at the kernel.
20205 * Therefore, the PM maintains a local list of timers for user processes
20206 * that requested an alarm signal.
20207 *
20208 * The entry points into this file are:
20209 * pm_set_timer:      reset and existing or set a new watchdog timer
20210 * pm_expire_timers: check for expired timers and run watchdog functions
20211 * pm_cancel_timer:   remove a time from the list of timers
20212 *
20213 */
20214
20215 #include "pm.h"
20216
20217 #include <timers.h>
20218 #include <minix/syslib.h>
20219 #include <minix/com.h>
20220
20221 PRIVATE timer_t *pm_timers = NULL;
20222
20223 /*=====
20224 *          pm_set_timer
20225 *=====
20226 PUBLIC void pm_set_timer(timer_t *tp, int ticks, tmr_func_t watchdog, int arg)
20227 {
20228     int r;
20229     clock_t now, prev_time = 0, next_time;
2030
2031     if ((r = getuptime(&now)) != OK)
2032         panic(__FILE__, "PM couldn't get uptime", NO_NUM);
2033
2034     /* Set timer argument and add timer to the list. */
2035     tmr_arg(tp)->ta_int = arg;
2036     prev_time = tmrs_settimer(&pm_timers, tp, now+ticks, watchdog, &next_time);
2037
2038     /* Reschedule our synchronous alarm if necessary. */
2039     if (! prev_time || prev_time > next_time) {
2040         if (sys_setalarm(next_time, 1) != OK)
```

```

20241             }
20242         }
20243     return;
20244 }
20245 }

20246 /*=====
20247 *          pm_expire_timers
20248 *=====
20249 PUBLIC void pm_expire_timers(clock_t now)
20250 {
20251     clock_t next_time;
20252
20253     /* Check for expired timers and possibly reschedule an alarm. */
20254     tmrs_exptimers(&pm_timers, now, &next_time);
20255     if (next_time > 0) {
20256         if (sys_setalarm(next_time, 1) != OK)
20257             panic(FILE__, "PM expire timer couldn't set alarm.", NO_NUM);
20258     }
20259 }
20260 }

20261 /*=====
20262 *          pm_cancel_timer
20263 *=====
20264 PUBLIC void pm_cancel_timer(timer_t *tp)
20265 {
20266     clock_t next_time, prev_time;
20267     prev_time = tmrs_clrtimer(&pm_timers, tp, &next_time);
20268
20269     /* If the earliest timer has been removed, we have to set the alarm to
20270      * the next timer, or cancel the alarm altogether if the last timer has
20271      * been cancelled (next_time will be 0 then).
20272      */
20273     if (prev_time < next_time || !next_time) {
20274         if (sys_setalarm(next_time, 1) != OK)
20275             panic(FILE__, "PM expire timer couldn't set alarm.", NO_NUM);
20276     }
20277 }
20278 }

+++++
servers/pm/time.c
+++++

20300 /* This file takes care of those system calls that deal with time.
20301 */
20302 /* The entry points into this file are
20303 *   do_time:           perform the TIME system call
20304 *   do_stime:          perform the STIME system call
20305 *   do_times:          perform the TIMES system call
20306 */
20307
20308 #include "pm.h"
20309 #include <minix/callnr.h>
20310 #include <minix/com.h>
20311 #include <signal.h>
20312 #include "mproc.h"
20313 #include "param.h"
20314
20315 PRIVATE time_t boottime;
20316
20317 /*=====
20318 *          do_time
20319 *=====
20320 PUBLIC int do_time()
20321 {
20322     /* Perform the time(tp) system call. This returns the time in seconds since

```

```

20323 * 1.1.1970. MINIX is an astrophysically naive system that assumes the earth
20324 * rotates at a constant rate and that such things as leap seconds do not
20325 * exist.
20326 */
20327 clock_t uptime;
20328 int s;
20329
20330 if ( (s=getuptime(&uptime)) != OK)
20331     panic(__FILE__,"do_time couldn't get uptime", s);
20332
20333 mp->mp_reply.reply_time = (time_t) (boottime + (uptime/HZ));
20334 mp->mp_reply.reply_utime = (uptime%HZ)*1000000/HZ;
20335 return(OK);
20336 }

20338 /*=====
20339 *          do_stime
20340 *=====
20341 PUBLIC int do_stime()
20342 {
20343 /* Perform the stime(tp) system call. Retrieve the system's uptime (ticks
20344 * since boot) and store the time in seconds at system boot in the global
20345 * variable 'boottime'.
20346 */
20347 clock_t uptime;
20348 int s;
20349
20350 if (mp->mp_effuid != SUPER_USER) {
20351     return(EPERM);
20352 }
20353 if ( (s=getuptime(&uptime)) != OK)
20354     panic(__FILE__,"do_stime couldn't get uptime", s);
20355 boottime = (long) m_in.stime - (uptime/HZ);
20356
20357 /* Also inform FS about the new system time. */
20358 tell_fs(STIME, boottime, 0, 0);
20359
20360 return(OK);
20361 }

20363 /*=====
20364 *          do_times
20365 *=====
20366 PUBLIC int do_times()
20367 {
20368 /* Perform the times(buffer) system call. */
20369 register struct mproc *rmp = mp;
20370 clock_t t[5];
20371 int s;
20372
20373 if (OK != (s=sys_times(who, t)))
20374     panic(__FILE__,"do_times couldn't get times", s);
20375 rmp->mp_reply.reply_t1 = t[0];           /* user time */
20376 rmp->mp_reply.reply_t2 = t[1];           /* system time */
20377 rmp->mp_reply.reply_t3 = rmp->mp_child_utime; /* child user time */
20378 rmp->mp_reply.reply_t4 = rmp->mp_child_stime; /* child system time */
20379 rmp->mp_reply.reply_t5 = t[4];           /* uptime since boot */
20380
20381 return(OK);
20382 }

+++++
servers/pm/getset.c
+++++
20400 /* This file handles the 4 system calls that get and set uids and gids.

```

```

20401 * It also handles getpid(), setsid(), and getpgrp(). The code for each
20402 * one is so tiny that it hardly seemed worthwhile to make each a separate
20403 * function.
20404 */
20405
20406 #include "pm.h"
20407 #include <minix/callnr.h>
20408 #include <signal.h>
20409 #include "mproc.h"
20410 #include "param.h"
20411
20412 /*=====
20413 *          do_getset
20414 *=====
20415 PUBLIC int do_getset()
20416 {
20417 /* Handle GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID. The four
20418 * GETs and SETSID return their primary results in 'r'. GETUID, GETGID, and
20419 * GETPID also return secondary results (the effective IDs, or the parent
20420 * process ID) in 'reply_res2', which is returned to the user.
20421 */
20422
20423 register struct mproc *rmp = mp;
20424 register int r;
20425
20426 switch(call_nr) {
20427     case GETUID:
20428         r = rmp->mp_realuid;
20429         rmp->mp_reply.reply_res2 = rmp->mp_effuid;
20430         break;
20431
20432     case GETGID:
20433         r = rmp->mp_realgid;
20434         rmp->mp_reply.reply_res2 = rmp->mp_effgid;
20435         break;
20436
20437     case GETPID:
20438         r = mproc[who].mp_pid;
20439         rmp->mp_reply.reply_res2 = mproc[rmp->mp_parent].mp_pid;
20440         break;
20441
20442     case SETUID:
20443         if (rmp->mp_realuid != (uid_t) m_in.usr_id &&
20444             rmp->mp_effuid != SUPER_USER)
20445             return(EPERM);
20446         rmp->mp_realuid = (uid_t) m_in.usr_id;
20447         rmp->mp_effuid = (uid_t) m_in.usr_id;
20448         tell_fs(SETUID, who, rmp->mp_realuid, rmp->mp_effuid);
20449         r = OK;
20450         break;
20451
20452     case SETGID:
20453         if (rmp->mp_realgid != (gid_t) m_in.grp_id &&
20454             rmp->mp_effuid != SUPER_USER)
20455             return(EPERM);
20456         rmp->mp_realgid = (gid_t) m_in.grp_id;
20457         rmp->mp_effgid = (gid_t) m_in.grp_id;
20458         tell_fs(SETGID, who, rmp->mp_realgid, rmp->mp_effgid);
20459         r = OK;
20460         break;
20461
20462     case SETSID:
20463         if (rmp->mp_proctyp == rmp->mp_pid) return(EPERM);
20464         rmp->mp_proctyp = rmp->mp_pid;
20465         tell_fs(SETSID, who, 0, 0);
20466         /* fall through */
20467

```

```

20468     case GETPGRP:
20469         r = rmp->mp_procgrp;
20470         break;
20471     default:
20472         r = EINVAL;
20473         break;
20474     }
20475 }
20476 return(r);
20477 }

+++++
servers/pm/misc.c
+++++

20500 /* Miscellaneous system calls.                                Author: Kees J. Bot
20501 *                                                               31 Mar 2000
20502 * The entry points into this file are:
20503 *   do_reboot: kill all processes, then reboot system
20504 *   do_svrcctl: process manager control
20505 *   do_getsysinfo: request copy of PM data structure (Jorrit N. Herder)
20506 *   do_getprocnr: lookup process slot number (Jorrit N. Herder)
20507 *   do_memalloc: allocate a chunk of memory (Jorrit N. Herder)
20508 *   do_memfree: deallocate a chunk of memory (Jorrit N. Herder)
20509 *   do_getsetpriority: get/set process priority
20510 */
20511
20512 #include "pm.h"
20513 #include <minix/callnr.h>
20514 #include <signal.h>
20515 #include <sys/svrcctl.h>
20516 #include <sys/resource.h>
20517 #include <minix/com.h>
20518 #include <string.h>
20519 #include "mproc.h"
20520 #include "param.h"
20521
20522 /*=====
20523 *          do_allocmem
20524 *=====
20525 PUBLIC int do_allocmem()
20526 {
20527     vir_clicks mem_clicks;
20528     phys_clicks mem_base;
20529
20530     mem_clicks = (m_in.memsize + CLICK_SIZE -1 ) >> CLICK_SHIFT;
20531     mem_base = alloc_mem(mem_clicks);
20532     if (mem_base == NO_MEM) return(ENOMEM);
20533     mp->mp_reply.membase = (phys_bytes) (mem_base << CLICK_SHIFT);
20534     return(OK);
20535 }

20536 /*=====
20537 *          do_freemem
20538 *=====
20539 PUBLIC int do_freemem()
20540 {
20541     vir_clicks mem_clicks;
20542     phys_clicks mem_base;
20543
20544     mem_clicks = (m_in.memsize + CLICK_SIZE -1 ) >> CLICK_SHIFT;
20545     mem_base = (m_in.membase + CLICK_SIZE -1 ) >> CLICK_SHIFT;
20546     free_mem(mem_base, mem_clicks);
20547     return(OK);
20548 }
20549 }
```

```

20551 /*=====
20552 *          do_getsysinfo
20553 *=====
20554 PUBLIC int do_getsysinfo()
20555 {
20556     struct mproc *proc_addr;
20557     vir_bytes src_addr, dst_addr;
20558     struct kinfo kinfo;
20559     size_t len;
20560     int s;
20561
20562     switch(m_in.info_what) {
20563         case SI_KINFO:           /* kernel info is obtained via PM */
20564             sys_getkinfo(&kinfo);
20565             src_addr = (vir_bytes) &kinfo;
20566             len = sizeof(struct kinfo);
20567             break;
20568         case SI_PROC_ADDR:       /* get address of PM process table */
20569             proc_addr = &mproc[0];
20570             src_addr = (vir_bytes) &proc_addr;
20571             len = sizeof(struct mproc *);
20572             break;
20573         case SI_PROC_TAB:        /* copy entire process table */
20574             src_addr = (vir_bytes) mproc;
20575             len = sizeof(struct mproc) * NR_PROCS;
20576             break;
20577         default:
20578             return(EINVAL);
20579     }
20580
20581     dst_addr = (vir_bytes) m_in.info_where;
20582     if (OK != (s=sys_datacopy(SELF, src_addr, who, dst_addr, len)))
20583         return(s);
20584     return(OK);
20585 }

20587 /*=====
20588 *          do_getprocnr
20589 *=====
20590 PUBLIC int do_getprocnr()
20591 {
20592     register struct mproc *rmp;
20593     static char search_key[PROC_NAME_LEN+1];
20594     int key_len;
20595     int s;
20596
20597     if (m_in.pid >= 0) {           /* lookup process by pid */
20598         for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
20599             if ((rmp->mp_flags & IN_USE) && (rmp->mp_pid==m_in.pid)) {
20600                 mp->mp_reply.procnr = (int) (rmp - mproc);
20601                 return(OK);
20602             }
20603         }
20604         return(ESRCH);
20605     } else if (m_in.namelen > 0) {   /* lookup process by name */
20606         key_len = MIN(m_in.namelen, PROC_NAME_LEN);
20607         if (OK != (s=sys_datacopy(who, (vir_bytes) m_in.addr,
20608                                     SELF, (vir_bytes) search_key, key_len)))
20609             return(s);
20610         search_key[key_len] = '\0';    /* terminate for safety */
20611         for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
20612             if ((rmp->mp_flags & IN_USE) &&
20613                 strncmp(rmp->mp_name, search_key, key_len)==0) {
20614                 mp->mp_reply.procnr = (int) (rmp - mproc);
20615                 return(OK);
20616             }
20617         }
20618     }
}

```

```

20618     return(ESRCH);
20619 } else {                                /* return own process number */
20620     mp->mp_reply.procnr = who;
20621 }
20622     return(OK);
20623 }

20625 /*=====
20626 *          do_reboot
20627 *=====
20628 #define REBOOT_CODE      "delay; boot"
20629 PUBLIC int do_reboot()
20630 {
20631     char monitor_code[32*sizeof(char *)];
20632     int code_len;
20633     int abort_flag;
20634
20635     if (mp->mp_effuid != SUPER_USER) return(EPERM);
20636
20637     switch (m_in.reboot_flag) {
20638     case RBT_HALT:
20639     case RBT_PANIC:
20640     case RBT_RESET:
20641         abort_flag = m_in.reboot_flag;
20642         break;
20643     case RBT_REBOOT:
20644         code_len = strlen(REBOOT_CODE) + 1;
20645         strncpy(monitor_code, REBOOT_CODE, code_len);
20646         abort_flag = RBT_MONITOR;
20647         break;
20648     case RBT_MONITOR:
20649         code_len = m_in.reboot_strlen + 1;
20650         if (code_len > sizeof(monitor_code)) return(EINVAL);
20651         if (sys_datacopy(who, (vir_bytes)m_in.reboot_code,
20652                         PM_PROC_NR, (vir_bytes)monitor_code,
20653                         (phys_bytes)(code_len)) != OK) return(EFAULT);
20654         if (monitor_code[code_len-1] != 0) return(EINVAL);
20655         abort_flag = RBT_MONITOR;
20656         break;
20657     default:
20658         return(EINVAL);
20659     }
20660
20661     check_sig(-1, SIGKILL);           /* kill all processes except init */
20662     tell_fs(REBOOT,0,0,0);           /* tell FS to prepare for shutdown */
20663
20664 /* Ask the kernel to abort. All system services, including the PM, will
20665 * get a HARD_STOP notification. Await the notification in the main loop.
20666 */
20667 sys_abort(abort_flag, PM_PROC_NR, monitor_code, code_len);
20668 return(SUSPEND);                      /* don't reply to killed process */
20669 }

20671 /*=====
20672 *          do_getsetpriority
20673 *=====
20674 PUBLIC int do_getsetpriority()
20675 {
20676     int arg_which, arg_who, arg_pri;
20677     int rmp_nr;
20678     struct mproc *rmp;
20679
20680     arg_which = m_in.m1_i1;
20681     arg_who = m_in.m1_i2;
20682     arg_pri = m_in.m1_i3; /* for SETPRIORITY */
20683
20684     /* Code common to GETPRIORITY and SETPRIORITY. */
20685

```

```

20686     /* Only support PRIO_PROCESS for now. */
20687     if (arg_which != PRIO_PROCESS)
20688         return(EINVAL);
20689
20690     if (arg_who == 0)
20691         rmp_nr = who;
20692     else
20693         if ((rmp_nr = proc_from_pid(arg_who)) < 0)
20694             return(ESRCH);
20695
20696     rmp = &mproc[rmp_nr];
20697
20698     if (mp->mp_effuid != SUPER_USER &&
20699         mp->mp_effuid != rmp->mp_effuid && mp->mp_effuid != rmp->mp_reuid)
20700         return EPERM;
20701
20702     /* If GET, that's it. */
20703     if (call_nr == GETPRIORITY) {
20704         return(rmp->mp_nice - PRIO_MIN);
20705     }
20706
20707     /* Only root is allowed to reduce the nice level. */
20708     if (rmp->mp_nice > arg_pri && mp->mp_effuid != SUPER_USER)
20709         return(EACCES);
20710
20711     /* We're SET, and it's allowed. Do it and tell kernel. */
20712     rmp->mp_nice = arg_pri;
20713     return sys_nice(rmp_nr, arg_pri);
20714 }

20715 /*=====
20716 *          do_svrcctl
20717 *=====
20718 */
20719 PUBLIC int do_svrcctl()
20720 {
20721     int s, req;
20722     vir_bytes ptr;
20723     #define MAX_LOCAL_PARAMS 2
20724     static struct {
20725         char name[30];
20726         char value[30];
20727     } local_param_overrides[MAX_LOCAL_PARAMS];
20728     static int local_params = 0;
20729
20730     req = m_in.svrcctl_req;
20731     ptr = (vir_bytes) m_in.svrcctl_argp;
20732
20733     /* Is the request indeed for the MM? */
20734     if (((req >> 8) & 0xFF) != 'M') return(EINVAL);
20735
20736     /* Control operations local to the PM. */
20737     switch(req) {
20738     case MMSETPARAM:
20739     case MMGETPARAM: {
20740         struct sysgetenv sysgetenv;
20741         char search_key[64];
20742         char *val_start;
20743         size_t val_len;
20744         size_t copy_len;
20745
20746         /* Copy sysgetenv structure to PM. */
20747         if (sys_datacopy(who, ptr, SELF, (vir_bytes) &sysgetenv,
20748                         sizeof(sysgetenv)) != OK) return(EFAULT);
20749
20750         /* Set a param override? */
20751         if (req == MMSETPARAM) {
20752             if (local_params >= MAX_LOCAL_PARAMS) return ENOSPC;

```

```

20753     if (sysgetenv.keylen <= 0
20754         || sysgetenv.keylen >=
20755             sizeof(local_param_overrides[local_params].name)
20756         || sysgetenv.vallen <= 0
20757         || sysgetenv.vallen >=
20758             sizeof(local_param_overrides[local_params].value))
20759         return EINVAL;
20760
20761     if ((s = sys_datacopy(who, (vir_bytes) sysgetenv.key,
20762         SELF, (vir_bytes) local_param_overrides[local_params].name,
20763         sysgetenv.keylen)) != OK)
20764         return s;
20765     if ((s = sys_datacopy(who, (vir_bytes) sysgetenv.val,
20766         SELF, (vir_bytes) local_param_overrides[local_params].value,
20767         sysgetenv.keylen)) != OK)
20768         return s;
20769     local_param_overrides[local_params].name[sysgetenv.keylen] = '\0';
20770     local_param_overrides[local_params].value[sysgetenv.vallen] = '\0';
20771
20772     local_params++;
20773
20774     return OK;
20775 }
20776
20777     if (sysgetenv.keylen == 0) { /* copy all parameters */
20778         val_start = monitor_params;
20779         val_len = sizeof(monitor_params);
20780     }
20781     else { /* lookup value for key */
20782         int p;
20783         /* Try to get a copy of the requested key. */
20784         if (sysgetenv.keylen > sizeof(search_key)) return(EINVAL);
20785         if ((s = sys_datacopy(who, (vir_bytes) sysgetenv.key,
20786             SELF, (vir_bytes) search_key, sysgetenv.keylen)) != OK)
20787             return(s);
20788
20789         /* Make sure key is null-terminated and lookup value.
20790         * First check local overrides.
20791         */
20792         search_key[sysgetenv.keylen-1] = '\0';
20793         for(p = 0; p < local_params; p++) {
20794             if (!strcmp(search_key, local_param_overrides[p].name)) {
20795                 val_start = local_param_overrides[p].value;
20796                 break;
20797             }
20798         }
20799         if (p >= local_params && (val_start = find_param(search_key)) == NULL)
20800             return(ESRCH);
20801         val_len = strlen(val_start) + 1;
20802     }
20803
20804     /* See if it fits in the client's buffer. */
20805     if (val_len > sysgetenv.vallen)
20806         return E2BIG;
20807
20808     /* Value found, make the actual copy (as far as possible). */
20809     copy_len = MIN(val_len, sysgetenv.vallen);
20810     if ((s=sys_datacopy(SELF, (vir_bytes) val_start,
20811         who, (vir_bytes) sysgetenv.val, copy_len)) != OK)
20812         return(s);
20813
20814     return OK;
20815 }
20816 default:
20817     return(EINVAL);
20818 }
20819 }
```

```
+++++
servers/fs/fs.h
+++++
20900 /* This is the master header for fs. It includes some other files
20901 * and defines the principal constants.
20902 */
20903 #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
20904 #define _MINIX             1    /* tell headers to include MINIX stuff */
20905 #define _SYSTEM            1    /* tell headers that this is the kernel */
20906
20907 #define VERBOSE           0    /* show messages during initialization? */
20908
20909 /* The following are so basic, all the *.c files get them automatically. */
20910 #include <minix/config.h>      /* MUST be first */
20911 #include <ansi.h>           /* MUST be second */
20912 #include <sys/types.h>
20913 #include <minix/const.h>
20914 #include <minix/type.h>
20915 #include <minix/dmap.h>
20916
20917 #include <limits.h>
20918 #include <errno.h>
20919
20920 #include <minix/syslib.h>
20921 #include <minix/sysutil.h>
20922
20923 #include "const.h"
20924 #include "type.h"
20925 #include "proto.h"
20926 #include "glo.h"
```

```
+++++
servers/fs/const.h
+++++
21000 /* Tables sizes */
21001 #define V1_NR_DZONES      7    /* # direct zone numbers in a V1 inode */
21002 #define V1_NR_TZONES       9    /* total # zone numbers in a V1 inode */
21003 #define V2_NR_DZONES      7    /* # direct zone numbers in a V2 inode */
21004 #define V2_NR_TZONES      10   /* total # zone numbers in a V2 inode */
21005
21006 #define NR_FILPS          128   /* # slots in filp table */
21007 #define NR_INODES          64    /* # slots in "in core" inode table */
21008 #define NR_SUPERS          8     /* # slots in super block table */
21009 #define NR_LOCKS           8     /* # slots in the file locking table */
21010
21011 /* The type of sizeof may be (unsigned) long. Use the following macro for
21012 * taking the sizes of small objects so that there are no surprises like
21013 * (small) long constants being passed to routines expecting an int.
21014 */
21015 #define usizeof(t) ((unsigned) sizeof(t))
21016
21017 /* File system types. */
21018 #define SUPER_MAGIC        0x137F /* magic number contained in super-block */
21019 #define SUPER_REV          0x7F13 /* magic # when 68000 disk read on PC or vv */
21020 #define SUPER_V2            0x2468 /* magic # for V2 file systems */
21021 #define SUPER_V2_REV        0x6824 /* V2 magic written on PC, read on 68K or vv */
21022 #define SUPER_V3            0x4d5a /* magic # for V3 file systems */
21023
21024 #define V1                  1    /* version number of V1 file systems */
21025 #define V2                  2    /* version number of V2 file systems */
21026 #define V3                  3    /* version number of V3 file systems */
21027
```

```

21028 /* Miscellaneous constants */
21029 #define SU_UID ((uid_t) 0) /* super_user's uid_t */
21030 #define SYS_UID ((uid_t) 0) /* uid_t for processes MM and INIT */
21031 #define SYS_GID ((gid_t) 0) /* gid_t for processes MM and INIT */
21032 #define NORMAL 0 /* forces get_block to do disk read */
21033 #define NO_READ 1 /* prevents get_block from doing disk read */
21034 #define PREFETCH 2 /* tells get_block not to read or mark dev */
21035
21036 #define XPIPE (-NR_TASKS-1) /* used in fp_task when susp'd on pipe */
21037 #define XLOCK (-NR_TASKS-2) /* used in fp_task when susp'd on lock */
21038 #define XPOPEN (-NR_TASKS-3) /* used in fp_task when susp'd on pipe open */
21039 #define XSELECT (-NR_TASKS-4) /* used in fp_task when susp'd on select */
21040
21041 #define NO_BIT ((bit_t) 0) /* returned by alloc_bit() to signal failure */
21042
21043 #define DUP_MASK 0100 /* mask to distinguish dup2 from dup */
21044
21045 #define LOOK_UP 0 /* tells search_dir to lookup string */
21046 #define ENTER 1 /* tells search_dir to make dir entry */
21047 #define DELETE 2 /* tells search_dir to delete entry */
21048 #define IS_EMPTY 3 /* tells search_dir to ret. OK or ENOTEMPTY */
21049
21050 #define CLEAN 0 /* disk and memory copies identical */
21051 #define DIRTY 1 /* disk and memory copies differ */
21052 #define ATIME 002 /* set if atime field needs updating */
21053 #define CTIME 004 /* set if ctime field needs updating */
21054 #define MTIME 010 /* set if mtime field needs updating */
21055
21056 #define BYTE_SWAP 0 /* tells conv2/conv4 to swap bytes */
21057
21058 #define END_OF_FILE (-104) /* eof detected */
21059
21060 #define ROOT_INODE 1 /* inode number for root directory */
21061 #define BOOT_BLOCK ((block_t) 0) /* block number of boot block */
21062 #define SUPER_BLOCK_BYTES (1024) /* bytes offset */
21063 #define START_BLOCK 2 /* first block of FS (not counting SB) */
21064
21065 #define DIR_ENTRY_SIZE usizeof (struct direct) /* # bytes/dir entry */
21066 #define NR_DIR_ENTRIES(b) ((b)/DIR_ENTRY_SIZE) /* # dir entries/blk */
21067 #define SUPER_SIZE usizeof (struct super_block) /* super_block size */
21068 #define PIPE_SIZE(b) (V1_NR_DZONES*(b)) /* pipe size in bytes */
21069
21070 #define FS_BITMAP_CHUNKS(b) ((b)/usizeof (bitchunk_t))/* # map chunks/blk */
21071 #define FS_BITCHUNK_BITS (usizeof(bitchunk_t) * CHAR_BIT)
21072 #define FS_BITS_PER_BLOCK(b) (FS_BITMAP_CHUNKS(b) * FS_BITCHUNK_BITS)
21073
21074 /* Derived sizes pertaining to the V1 file system. */
21075 #define V1_ZONE_NUM_SIZE usizeof (zone1_t) /* # bytes in V1 zone */
21076 #define V1_INODE_SIZE usizeof (d1_inode) /* bytes in V1 dsk ino */
21077
21078 /* # zones/indir block */
21079 #define V1_INDIRECTS (STATIC_BLOCK_SIZE/V1_ZONE_NUM_SIZE)
21080
21081 /* # V1 dsk inodes/blk */
21082 #define V1_INODES_PER_BLOCK (STATIC_BLOCK_SIZE/V1_INODE_SIZE)
21083
21084 /* Derived sizes pertaining to the V2 file system. */
21085 #define V2_ZONE_NUM_SIZE usizeof (zone2_t) /* # bytes in V2 zone */
21086 #define V2_INODE_SIZE usizeof (d2_inode) /* bytes in V2 dsk ino */
21087 #define V2_INDIRECTS(b) ((b)/V2_ZONE_NUM_SIZE) /* # zones/indir block */
21088 #define V2_INODES_PER_BLOCK(b) ((b)/V2_INODE_SIZE)/* # V2 dsk inodes/blk */

+++++
servers/fs/type.h
+++++

```

21100 /\* Declaration of the V1 inode as it is on the disk (not in core). \*/

```

21101 typedef struct { /* V1.x disk inode */
21102     mode_t d1_mode; /* file type, protection, etc. */
21103     uid_t d1_uid; /* user id of the file's owner */
21104     off_t d1_size; /* current file size in bytes */
21105     time_t d1_mtime; /* when was file data last changed */
21106     u8_t d1_gid; /* group number */
21107     u8_t d1_nlinks; /* how many links to this file */
21108     u16_t d1_zone[V1_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
21109 } d1_inode;
21110
21111 /* Declaration of the V2 inode as it is on the disk (not in core). */
21112 typedef struct { /* V2.x disk inode */
21113     mode_t d2_mode; /* file type, protection, etc. */
21114     u16_t d2_nlinks; /* how many links to this file. HACK! */
21115     uid_t d2_uid; /* user id of the file's owner. */
21116     u16_t d2_gid; /* group number HACK! */
21117     off_t d2_size; /* current file size in bytes */
21118     time_t d2_atime; /* when was file data last accessed */
21119     time_t d2_mtime; /* when was file data last changed */
21120     time_t d2_ctime; /* when was inode data last changed */
21121     zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
21122 } d2_inode;

```

```
+++++
servers/fs/proto.h
+++++
```

```

21200 /* Function prototypes. */
21201
21202 #include "timers.h"
21203
21204 /* Structs used in prototypes must be declared as such first. */
21205 struct buf;
21206 struct filp;
21207 struct inode;
21208 struct super_block;
21209
21210 /* cache.c */
21211 _PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z) );
21212 _PROTOTYPE( void flushall, (Dev_t dev) );
21213 _PROTOTYPE( void free_zone, (Dev_t dev, zone_t numb) );
21214 _PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block, int only_search));
21215 _PROTOTYPE( void invalidate, (Dev_t device) );
21216 _PROTOTYPE( void put_block, (struct buf *bp, int block_type) );
21217 _PROTOTYPE( void rw_block, (struct buf *bp, int rw_flag) );
21218 _PROTOTYPE( void rw_scattered, (Dev_t dev,
21219             struct buf **bufq, int bufqsize, int rw_flag) );
21220
21221 /* device.c */
21222 _PROTOTYPE( int dev_open, (Dev_t dev, int proc, int flags) );
21223 _PROTOTYPE( void dev_close, (Dev_t dev) );
21224 _PROTOTYPE( int dev_io, (int op, Dev_t dev, int proc, void *buf,
21225             off_t pos, int bytes, int flags) );
21226 _PROTOTYPE( int gen_opcl, (int op, Dev_t dev, int proc, int flags) );
21227 _PROTOTYPE( void gen_io, (int task_nr, message *mess_ptr) );
21228 _PROTOTYPE( int no_dev, (int op, Dev_t dev, int proc, int flags) );
21229 _PROTOTYPE( int tty_opcl, (int op, Dev_t dev, int proc, int flags) );
21230 _PROTOTYPE( int ctty_opcl, (int op, Dev_t dev, int proc, int flags) );
21231 _PROTOTYPE( int clone_opcl, (int op, Dev_t dev, int proc, int flags) );
21232 _PROTOTYPE( void ctty_io, (int task_nr, message *mess_ptr) );
21233 _PROTOTYPE( int do_ioctl, (void) );
21234 _PROTOTYPE( int do_setsid, (void) );
21235 _PROTOTYPE( void dev_status, (message *) );
21236
21237 /* dmp.c */

```

```

21238 _PROTOTYPE( int do_fkey_pressed, (void) );  

21239  

21240 /* dmap.c */  

21241 _PROTOTYPE( int do_devctl, (void) );  

21242 _PROTOTYPE( void build_dmap, (void) );  

21243 _PROTOTYPE( int map_driver, (int major, int proc_nr, int dev_style) );  

21244  

21245 /* filedes.c */  

21246 _PROTOTYPE( struct filp *find_filp, (struct inode *rip, mode_t bits) );  

21247 _PROTOTYPE( int get_fd, (int start, mode_t bits, int *k, struct filp **fpt) );  

21248 _PROTOTYPE( struct filp *get_filp, (int fild) );  

21249  

21250 /* inode.c */  

21251 _PROTOTYPE( struct inode *alloc_inode, (dev_t dev, mode_t bits) );  

21252 _PROTOTYPE( void dup_inode, (struct inode *ip) );  

21253 _PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb) );  

21254 _PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb) );  

21255 _PROTOTYPE( void put_inode, (struct inode *rip) );  

21256 _PROTOTYPE( void update_times, (struct inode *rip) );  

21257 _PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag) );  

21258 _PROTOTYPE( void wipe_inode, (struct inode *rip) );  

21259  

21260 /* link.c */  

21261 _PROTOTYPE( int do_link, (void) );  

21262 _PROTOTYPE( int do_unlink, (void) );  

21263 _PROTOTYPE( int do_rename, (void) );  

21264 _PROTOTYPE( void truncate, (struct inode *rip) );  

21265  

21266 /* lock.c */  

21267 _PROTOTYPE( int lock_op, (struct filp *f, int req) );  

21268 _PROTOTYPE( void lock_revive, (void) );  

21269  

21270 /* main.c */  

21271 _PROTOTYPE( int main, (void) );  

21272 _PROTOTYPE( void reply, (int whom, int result) );  

21273  

21274 /* misc.c */  

21275 _PROTOTYPE( int do_dup, (void) );  

21276 _PROTOTYPE( int do_exit, (void) );  

21277 _PROTOTYPE( int do_fcntl, (void) );  

21278 _PROTOTYPE( int do_fork, (void) );  

21279 _PROTOTYPE( int do_exec, (void) );  

21280 _PROTOTYPE( int do_revive, (void) );  

21281 _PROTOTYPE( int do_set, (void) );  

21282 _PROTOTYPE( int do_sync, (void) );  

21283 _PROTOTYPE( int do_fsync, (void) );  

21284 _PROTOTYPE( int do_reboot, (void) );  

21285 _PROTOTYPE( int do_svrcntl, (void) );  

21286 _PROTOTYPE( int do_getsysinfo, (void) );  

21287  

21288 /* mount.c */  

21289 _PROTOTYPE( int do_mount, (void) );  

21290 _PROTOTYPE( int do_umount, (void) );  

21291 _PROTOTYPE( int unmount, (Dev_t dev) );  

21292  

21293 /* open.c */  

21294 _PROTOTYPE( int do_close, (void) );  

21295 _PROTOTYPE( int do_creat, (void) );  

21296 _PROTOTYPE( int do_lseek, (void) );  

21297 _PROTOTYPE( int do_mknod, (void) );  

21298 _PROTOTYPE( int do_mkdir, (void) );  

21299 _PROTOTYPE( int do_open, (void) );  

21300  

21301 /* path.c */  

21302 _PROTOTYPE( struct inode *advance,(struct inode *dirp, char string[NAME_MAX]));  

21303 _PROTOTYPE( int search_dir, (struct inode *ldir_ptr,  

21304           char string [NAME_MAX], ino_t *numb, int flag) );

```

```

21305 _PROTOTYPE( struct inode *eat_path, (char *path) );
21306 _PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX])); 
21307
21308 /* pipe.c */
21309 _PROTOTYPE( int do_pipe, (void) );
21310 _PROTOTYPE( int do_unpause, (void) );
21311 _PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
21312         int oflags, int bytes, off_t position, int *canwrite, int notouch));
21313 _PROTOTYPE( void release, (struct inode *ip, int call_nr, int count) );
21314 _PROTOTYPE( void revive, (int proc_nr, int bytes) );
21315 _PROTOTYPE( void suspend, (int task) );
21316 _PROTOTYPE( int select_request_pipe, (struct filp *f, int *ops, int bl) );
21317 _PROTOTYPE( int select_cancel_pipe, (struct filp *f) );
21318 _PROTOTYPE( int select_match_pipe, (struct filp *f) );
21319
21320 /* protect.c */
21321 _PROTOTYPE( int do_access, (void) );
21322 _PROTOTYPE( int do_chmod, (void) );
21323 _PROTOTYPE( int do_chown, (void) );
21324 _PROTOTYPE( int do_umask, (void) );
21325 _PROTOTYPE( int forbidden, (struct inode *rip, mode_t access_desired) );
21326 _PROTOTYPE( int read_only, (struct inode *ip) );
21327
21328 /* read.c */
21329 _PROTOTYPE( int do_read, (void) );
21330 _PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
21331             off_t position, unsigned bytes_ahead) );
21332 _PROTOTYPE( void read_ahead, (void) );
21333 _PROTOTYPE( block_t read_map, (struct inode *rip, off_t position) );
21334 _PROTOTYPE( int read_write, (int rw_flag) );
21335 _PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index) );
21336
21337 /* statir.c */
21338 _PROTOTYPE( int do_chdir, (void) );
21339 _PROTOTYPE( int do_fchdir, (void) );
21340 _PROTOTYPE( int do_chroot, (void) );
21341 _PROTOTYPE( int do_fstat, (void) );
21342 _PROTOTYPE( int do_stat, (void) );
21343 _PROTOTYPE( int do_fstatfs, (void) );
21344
21345 /* super.c */
21346 _PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
21347 _PROTOTYPE( void free_bit, (struct super_block *sp, int map,
21348                 bit_t bit_returned) );
21349 _PROTOTYPE( struct super_block *get_super, (Dev_t dev) );
21350 _PROTOTYPE( int mounted, (struct inode *rip) );
21351 _PROTOTYPE( int read_super, (struct super_block *sp) );
21352 _PROTOTYPE( int get_block_size, (dev_t dev) );
21353
21354 /* time.c */
21355 _PROTOTYPE( int do_stime, (void) );
21356 _PROTOTYPE( int do_utime, (void) );
21357
21358 /* utility.c */
21359 _PROTOTYPE( time_t clock_time, (void) );
21360 _PROTOTYPE( unsigned conv2, (int norm, int w) );
21361 _PROTOTYPE( long conv4, (int norm, long x) );
21362 _PROTOTYPE( int fetch_name, (char *path, int len, int flag) );
21363 _PROTOTYPE( int no_sys, (void) );
21364 _PROTOTYPE( void panic, (char *who, char *mess, int num) );
21365
21366 /* write.c */
21367 _PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag) );
21368 _PROTOTYPE( int do_write, (void) );
21369 _PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position) );
21370 _PROTOTYPE( void zero_block, (struct buf *bp) );
21371

```

```

21372 /* select.c */
21373 _PROTOTYPE( int do_select, (void) );
21374 _PROTOTYPE( int select_callback, (struct filp *, int ops) );
21375 _PROTOTYPE( void select_forget, (int fproc) );
21376 _PROTOTYPE( void select_timeout_check, (timer_t *) );
21377 _PROTOTYPE( void init_select, (void) );
21378 _PROTOTYPE( int select_notified, (int major, int minor, int ops) );
21379
21380 /* timers.c */
21381 _PROTOTYPE( void fs_set_timer, (timer_t *tp, int delta, tmr_func_t watchdog, int arg));
21382 _PROTOTYPE( void fs_expire_timers, (clock_t now) );
21383 _PROTOTYPE( void fs_cancel_timer, (timer_t *tp) );
21384 _PROTOTYPE( void fs_init_timer, (timer_t *tp) );
21385
21386 /* cdprobe.c */
21387 _PROTOTYPE( int cdprobe, (void) );

```

```
+++++
servers/fs/glo.h
+++++
```

```

21400 /* EXTERN should be extern except for the table file */
21401 #ifdef _TABLE
21402 #undef EXTERN
21403 #define EXTERN
21404 #endif
21405
21406 /* File System global variables */
21407 EXTERN struct fproc *fp;           /* pointer to caller's fproc struct */
21408 EXTERN int super_user;           /* 1 if caller is super_user, else 0 */
21409 EXTERN int susp_count;          /* number of procs suspended on pipe */
21410 EXTERN int nr_locks;            /* number of locks currently in place */
21411 EXTERN int reviving;           /* number of pipe processes to be revived */
21412 EXTERN off_t rdahedpos;        /* position to read ahead */
21413 EXTERN struct inode *rdahed_inode; /* pointer to inode to read ahead */
21414 EXTERN Dev_t root_dev;         /* device number of the root device */
21415 EXTERN time_t boottime;        /* time in seconds at system boot */
21416
21417 /* The parameters of the call are kept here. */
21418 EXTERN message m_in;           /* the input message itself */
21419 EXTERN message m_out;          /* the output message used for reply */
21420 EXTERN int who;               /* caller's proc number */
21421 EXTERN int call_nr;           /* system call number */
21422 EXTERN char user_path[PATH_MAX];/* storage for user path name */
21423
21424 /* The following variables are used for returning results to the caller. */
21425 EXTERN int err_code;           /* temporary storage for error number */
21426 EXTERN int rdwt_err;           /* status of last disk i/o request */
21427
21428 /* Data initialized elsewhere. */
21429 extern _PROTOTYPE( int (*call_vec[]), (void) ); /* sys call table */
21430 extern char dot1[2];           /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
21431 extern char dot2[3];           /* meaning to search_dir: no access permission check. */

```

```
+++++
servers/fs/fproc.h
+++++
```

```

21500 /* This is the per-process information. A slot is reserved for each potential
21501 * process. Thus NR_PROCS must be the same as in the kernel. It is not
21502 * possible or even necessary to tell when a slot is free here.
21503 */

```

```

21504 EXTERN struct fproc {
21505     mode_t fp_umask;           /* mask set by umask system call */
21506     struct inode *fp_workdir;   /* pointer to working directory's inode */
21507     struct inode *fp_rootdir;   /* pointer to current root dir (see chroot) */
21508     struct filp *fp_filp[OPEN_MAX];/* the file descriptor table */
21509     uid_t fp_reuid;           /* real user id */
21510     uid_t fp_effuid;          /* effective user id */
21511     gid_t fp_realgid;         /* real group id */
21512     gid_t fp_effgid;          /* effective group id */
21513     dev_t fp_tty;             /* major/minor of controlling tty */
21514     int fp_fd;                /* place to save fd if rd/wr can't finish */
21515     char *fp_buffer;           /* place to save buffer if rd/wr can't finish */
21516     int fp_nbytes;             /* place to save bytes if rd/wr can't finish */
21517     int fp_cum_io_partial;    /* partial byte count if rd/wr can't finish */
21518     char fp_suspended;        /* set to indicate process hanging */
21519     char fp_revived;          /* set to indicate process being revived */
21520     char fp_task;              /* which task is proc suspended on */
21521     char fp_sesldr;            /* true if proc is a session leader */
21522     pid_t fp_pid;              /* process id */
21523     long fp_cloexec;           /* bit map for POSIX Table 6-2 FD_CLOEXEC */
21524 } fproc[NR_PROCS];
21525
21526 /* Field values. */
21527 #define NOT_SUSPENDED      0 /* process is not suspended on pipe or task */
21528 #define SUSPENDED          1 /* process is suspended on pipe or task */
21529 #define NOT_REVIVING        0 /* process is not being revived */
21530 #define REVIVING            1 /* process is being revived from suspension */
21531 #define PID_FREE            0 /* process slot free */
21532
21533 /* Check is process number is acceptable - includes system processes. */
21534 #define isokprocnr(n) ((unsigned)((n)+NR_TASKS) < NR_PROCS + NR_TASKS)
21535

```

```
+++++
servers/fs/buf.h
+++++
```

```

21600 /* Buffer (block) cache. To acquire a block, a routine calls get_block(),
21601 * telling which block it wants. The block is then regarded as "in use"
21602 * and has its 'b_count' field incremented. All the blocks that are not
21603 * in use are chained together in an LRU list, with 'front' pointing
21604 * to the least recently used block, and 'rear' to the most recently used
21605 * block. A reverse chain, using the field b_prev is also maintained.
21606 * Usage for LRU is measured by the time the put_block() is done. The second
21607 * parameter to put_block() can violate the LRU order and put a block on the
21608 * front of the list, if it will probably not be needed soon. If a block
21609 * is modified, the modifying routine must set b_dirt to DIRTY, so the block
21610 * will eventually be rewritten to the disk.
21611 */
21612
21613 #include <sys/dir.h>                      /* need struct direct */
21614 #include <dirent.h>
21615
21616 EXTERN struct buf {
21617     /* Data portion of the buffer. */
21618     union {
21619         char b_data[MAX_BLOCK_SIZE];           /* ordinary user data */
21620     /* directory block */
21621         struct direct b_dir[NR_DIR_ENTRIES(MAX_BLOCK_SIZE)];
21622     /* V1 indirect block */
21623         zone1_t b_v1_ind[V1_INDIRECTS];
21624     /* V2 indirect block */
21625         zone_t b_v2_ind[V2_INDIRECTS(MAX_BLOCK_SIZE)];

```

```

21626 /* V1 inode block */
21627     d1_inode b_v1_ino[V1_INODES_PER_BLOCK];
21628 /* V2 inode block */
21629     d2_inode b_v2_ino[V2_INODES_PER_BLOCK(MAX_BLOCK_SIZE)];
21630 /* bit map block */
21631     bitchunk_t b_bitmap[FS_BITMAP_CHUNKS(MAX_BLOCK_SIZE)];
21632 } b;
21633
21634 /* Header portion of the buffer. */
21635 struct buf *b_next;           /* used to link all free bufs in a chain */
21636 struct buf *b_prev;          /* used to link all free bufs the other way */
21637 struct buf *b_hash;          /* used to link bufs on hash chains */
21638 block_t b_blocknr;          /* block number of its (minor) device */
21639 dev_t b_dev;               /* major | minor device where block resides */
21640 char b_dirt;               /* CLEAN or DIRTY */
21641 char b_count;              /* number of users of this buffer */
21642 } buf[NR_BUFS];
21643
21644 /* A block is free if b_dev == NO_DEV. */
21645
21646 #define NIL_BUF ((struct buf *) 0)      /* indicates absence of a buffer */
21647
21648 /* These defs make it possible to use to bp->b_data instead of bp->b.b_data */
21649 #define b_data   b.b_data
21650 #define b_dir    b.b_dir
21651 #define b_v1_ind b.b_v1_ind
21652 #define b_v2_ind b.b_v2_ind
21653 #define b_v1_ino b.b_v1_ino
21654 #define b_v2_ino b.b_v2_ino
21655 #define b_bitmap b.b_bitmap
21656
21657 EXTERN struct buf *buf_hash[NR_BUF_HASH];      /* the buffer hash table */
21658
21659 EXTERN struct buf *front;           /* points to least recently used free block */
21660 EXTERN struct buf *rear;            /* points to most recently used free block */
21661 EXTERN int bufs_in_use;           /* # bufs currently in use (not on free list)*/
21662
21663 /* When a block is released, the type of usage is passed to put_block(). */
21664 #define WRITE_IMMED 0100 /* block should be written to disk now */
21665 #define ONE_SHOT    0200 /* set if block not likely to be needed soon */
21666
21667 #define INODE_BLOCK    0           /* inode block */
21668 #define DIRECTORY_BLOCK 1         /* directory block */
21669 #define INDIRECT_BLOCK 2         /* pointer block */
21670 #define MAP_BLOCK      3           /* bit map */
21671 #define FULL_DATA_BLOCK 5         /* data, fully used */
21672 #define PARTIAL_DATA_BLOCK 6        /* data, partly used*/
21673
21674 #define HASH_MASK (NR_BUF_HASH - 1) /* mask for hashing block numbers */

+++++
servers/fs/file.h
+++++
21700 /* This is the filp table. It is an intermediary between file descriptors and
21701 * inodes. A slot is free if filp_count == 0.
21702 */
21703
21704 EXTERN struct filp {
21705     mode_t filp_mode;           /* RW bits, telling how file is opened */
21706     int filp_flags;             /* Flags from open and fcntl */
21707     int filp_count;             /* how many file descriptors share this slot? */
21708     struct inode *filp_ino;     /* pointer to the inode */
21709     off_t filp_pos;             /* file position */

```

```

21710
21711 /* the following fields are for select() and are owned by the generic
21712 * select() code (i.e., fd-type-specific select() code can't touch these).
21713 */
21714 int filp_selectors;           /* select()ing processes blocking on this fd */
21715 int filp_select_ops;         /* interested in these SEL_* operations */
21716
21717 /* following are for fd-type-specific select() */
21718 int filp_pipe_select_ops;
21719 } filp[NR_FILPS];
21720
21721 #define FILP_CLOSED      0      /* filp_mode: associated device closed */
21722
21723 #define NIL_FILP (struct filp *) 0      /* indicates absence of a filp slot */

```

```
+++++
servers/fs/lock.h
+++++
```

```

21800 /* This is the file locking table. Like the filp table, it points to the
21801 * inode table, however, in this case to achieve advisory locking.
21802 */
21803 EXTERN struct file_lock {
21804     short lock_type;          /* F_RDLCK or F_WRLCK; 0 means unused slot */
21805     pid_t lock_pid;          /* pid of the process holding the lock */
21806     struct inode *lock_inode; /* pointer to the inode locked */
21807     off_t lock_first;        /* offset of first byte locked */
21808     off_t lock_last;         /* offset of last byte locked */
21809 } file_lock[NR_LOCKS];

```

```
+++++
servers/fs/inode.h
+++++
```

```

21900 /* Inode table. This table holds inodes that are currently in use. In some
21901 * cases they have been opened by an open() or creat() system call, in other
21902 * cases the file system itself needs the inode for one reason or another,
21903 * such as to search a directory for a path name.
21904 * The first part of the struct holds fields that are present on the
21905 * disk; the second part holds fields not present on the disk.
21906 * The disk inode part is also declared in "type.h" as 'd1_inode' for V1
21907 * file systems and 'd2_inode' for V2 file systems.
21908 */
21909
21910 EXTERN struct inode {
21911     mode_t i_mode;            /* file type, protection, etc. */
21912     nlink_t i_nlinks;         /* how many links to this file */
21913     uid_t i_uid;              /* user id of the file's owner */
21914     gid_t i_gid;              /* group number */
21915     off_t i_size;             /* current file size in bytes */
21916     time_t i_atime;           /* time of last access (V2 only) */
21917     time_t i_mtime;           /* when was file data last changed */
21918     time_t i_ctime;           /* when was inode itself changed (V2 only) */
21919     zone_t i_zone[N2_NR_TZONES]; /* zone numbers for direct, ind, and dbl ind */
21920
21921 /* The following items are not present on the disk. */
21922     dev_t i_dev;               /* which device is the inode on */
21923     ino_t i_num;               /* inode number on its (minor) device */
21924     int i_count;                /* # times inode used; 0 means slot is free */
21925     int i_ndzones;              /* # direct zones (Vx_NR_DZONES) */
21926     int i_nindirs;              /* # indirect zones per indirect block */
21927     struct super_block *i_sp;    /* pointer to super block for inode's device */
21928     char i_dirt;                /* CLEAN or DIRTY */
21929     char i_pipe;                /* set to I_PIPE if pipe */
21930     char i_mount;                /* this bit is set if file mounted on */
21931     char i_seek;                /* set on LSEEK, cleared on READ/WRITE */
21932     char i_update;               /* the ATIME, CTIME, and MTIME bits are here */

```

```

21933 } inode[NR_INODES];
21934
21935 #define NIL_INODE (struct inode *) 0 /* indicates absence of inode slot */
21936
21937 /* Field values. Note that CLEAN and DIRTY are defined in "const.h" */
21938 #define NO_PIPE 0 /* i_pipe is NO_PIPE if inode is not a pipe */
21939 #define I_PIPE 1 /* i_pipe is I_PIPE if inode is a pipe */
21940 #define NO_MOUNT 0 /* i_mount is NO_MOUNT if file not mounted on*/
21941 #define I_MOUNT 1 /* i_mount is I_MOUNT if file mounted on */
21942 #define NO_SEEK 0 /* i_seek = NO_SEEK if last op was not SEEK */
21943 #define ISEEK 1 /* i_seek = ISEEK if last op was SEEK */

+++++
servers/fs/param.h
+++++

22000 /* The following names are synonyms for the variables in the input message. */
22001 #define acc_time m2_ll
22002 #define addr m1_i3
22003 #define buffer m1_p1
22004 #define child m1_i2
22005 #define co_mode m1_i1
22006 #define eff_grp_id m1_i3
22007 #define eff_user_id m1_i3
22008 #define erki m1_p1
22009 #define fd m1_i1
22010 #define fd2 m1_i2
22011 #define ioflags m1_i3
22012 #define group m1_i3
22013 #define real_grp_id m1_i2
22014 #define ls_fd m2_i1
22015 #define mk_mode m1_i2
22016 #define mk_z0 m1_i3
22017 #define mode m3_i2
22018 #define c_mode m1_i3
22019 #define c_name m1_p1
22020 #define name m3_p1
22021 #define name1 m1_p1
22022 #define name2 m1_p2
22023 #define name_length m3_i1
22024 #define name1_length m1_i1
22025 #define name2_length m1_i2
22026 #define nbytes m1_i2
22027 #define owner m1_i2
22028 #define parent m1_i1
22029 #define pathname m3_cal
22030 #define pid m1_i3
22031 #define pro m1_i1
22032 #define ctl_req m4_ll
22033 #define driver_nr m4_l2
22034 #define dev_nr m4_l3
22035 #define dev_style m4_l4
22036 #define rd_only m1_i3
22037 #define real_user_id m1_i2
22038 #define request m1_i2
22039 #define sig m1_i2
22040 #define slot1 m1_i1
22041 #define tp m2_ll
22042 #define utime_actime m2_ll
22043 #define utime_modtime m2_l2
22044 #define utime_file m2_p1
22045 #define utime_length m2_i1
22046 #define utime_strlen m2_i2
22047 #define whence m2_i2
22048 #define svrctl_req m2_i1
22049 #define svrctl_argp m2_p1

```

```

22050 #define pm_stime      m1_i1
22051 #define info_what    m1_i1
22052 #define info_where   m1_p1
22053
22054 /* The following names are synonyms for the variables in the output message. */
22055 #define reply_type    m_type
22056 #define reply_l1       m2_l1
22057 #define reply_i1       m1_i1
22058 #define reply_i2       m1_i2
22059 #define reply_t1       m4_l1
22060 #define reply_t2       m4_i2
22061 #define reply_t3       m4_l3
22062 #define reply_t4       m4_l4
22063 #define reply_t5       m4_l5

+++++servers/fs/super.h+++++
22100 /* Super block table. The root file system and every mounted file system
22101 * has an entry here. The entry holds information about the sizes of the bit
22102 * maps and inodes. The s_ninodes field gives the number of inodes available
22103 * for files and directories, including the root directory. Inode 0 is
22104 * on the disk, but not used. Thus s_ninodes = 4 means that 5 bits will be
22105 * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
22106 * for files and directories. The disk layout is:
22107 *
22108 *     Item          # blocks
22109 *     boot block    1
22110 *     super block   1    (offset 1kB)
22111 *     inode map     s_imap_blocks
22112 *     zone map      s_zmap_blocks
22113 *     inodes        (s_ninodes + 'inodes per block' - 1) / 'inodes per block'
22114 *     unused         whatever is needed to fill out the current zone
22115 *     data zones    (s_zones - s_firstdatazone) << s_log_zone_size
22116 *
22117 * A super_block slot is free if s_dev == NO_DEV.
22118 */
22119
22120 EXTERN struct super_block {
22121     ino_t s_ninodes;           /* # usable inodes on the minor device */
22122     zone1_t s_nzones;         /* total device size, including bit maps etc */
22123     short s_imap_blocks;     /* # of blocks used by inode bit map */
22124     short s_zmap_blocks;     /* # of blocks used by zone bit map */
22125     zone1_t s_firstdatazone; /* number of first data zone */
22126     short s_log_zone_size;   /* log2 of blocks/zone */
22127     short s_pad;             /* try to avoid compiler-dependent padding */
22128     off_t s_max_size;        /* maximum file size on this device */
22129     zone_t s_zones;          /* number of zones (replaces s_nzones in V2) */
22130     short s_magic;           /* magic number to recognize super-blocks */
22131
22132 /* The following items are valid on disk only for V3 and above */
22133
22134 /* The block size in bytes. Minimum MIN_BLOCK_SIZE. SECTOR_SIZE
22135 * multiple. If V1 or V2 filesystem, this should be
22136 * initialised to STATIC_BLOCK_SIZE. Maximum MAX_BLOCK_SIZE.
22137 */
22138 short s_pad2;                /* try to avoid compiler-dependent padding */
22139 unsigned short s_block_size; /* block size in bytes. */
22140 char s_disk_version;        /* filesystem format sub-version */
22141
22142 /* The following items are only used when the super_block is in memory. */
22143 struct inode *s_isup;        /* inode for root dir of mounted file sys */
22144 struct inode *s_imount;      /* inode mounted on */
22145 unsigned s_inodes_per_block; /* precalculated from magic number */
22146 dev_t s_dev;                /* whose super block is this? */

```

```

22147     int s_rd_only;           /* set to 1 iff file sys mounted read only */
22148     int s_native;          /* set to 1 iff not byte swapped file system */
22149     int s_version;          /* file system version, zero means bad magic */
22150     int s_ndzones;          /* # direct zones in an inode */
22151     int s_nindirs;          /* # indirect zones per indirect block */
22152     bit_t s_isearch;        /* inodes below this bit number are in use */
22153     bit_t s_zsearch;        /* all zones below this bit number are in use*/
22154 } super_block[NR_SUPERS];
22155
22156 #define NIL_SUPER (struct super_block *) 0
22157 #define IMAP      0          /* operating on the inode bit map */
22158 #define ZMAP      1          /* operating on the zone bit map */

+++++
servers/fs/table.c
+++++
22200 /* This file contains the table used to map system call numbers onto the
22201 * routines that perform them.
22202 */
22203
22204 #define _TABLE
22205
22206 #include "fs.h"
22207 #include <minix/callnr.h>
22208 #include <minix/com.h>
22209 #include "buf.h"
22210 #include "file.h"
22211 #include "fproc.h"
22212 #include "inode.h"
22213 #include "lock.h"
22214 #include "super.h"
22215
22216 PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
22217     no_sys,           /* 0 = unused */
22218     do_exit,          /* 1 = exit */
22219     do_fork,          /* 2 = fork */
22220     do_read,          /* 3 = read */
22221     do_write,         /* 4 = write */
22222     do_open,          /* 5 = open */
22223     do_close,         /* 6 = close */
22224     no_sys,           /* 7 = wait */
22225     do_creat,         /* 8 = creat */
22226     do_link,          /* 9 = link */
22227     do_unlink,        /* 10 = unlink */
22228     no_sys,           /* 11 = waitpid */
22229     do_chdir,         /* 12 = chdir */
22230     no_sys,           /* 13 = time */
22231     do_mknod,         /* 14 = mknod */
22232     do_chmod,         /* 15 = chmod */
22233     do_chown,         /* 16 = chown */
22234     no_sys,           /* 17 = break */
22235     do_stat,          /* 18 = stat */
22236     do_lseek,         /* 19 = lseek */
22237     no_sys,           /* 20 = getpid */
22238     do_mount,         /* 21 = mount */
22239     do_umount,        /* 22 = umount */
22240     do_set,           /* 23 = setuid */
22241     no_sys,           /* 24 = getuid */
22242     do_stime,         /* 25 = stime */
22243     no_sys,           /* 26 = ptrace */
22244     no_sys,           /* 27 = alarm */
22245     do_fstat,         /* 28 = fstat */
22246     no_sys,           /* 29 = pause */
22247     do_utime,         /* 30 = utime */
22248     no_sys,           /* 31 = (stty) */

```

```

22249      no_sys,          /* 32 = (gtty) */
22250      do_access,       /* 33 = access */
22251      no_sys,          /* 34 = (nice) */
22252      no_sys,          /* 35 = (ftime) */
22253      do_sync,          /* 36 = sync */
22254      no_sys,          /* 37 = kill */
22255      do_rename,        /* 38 = rename */
22256      do_mkdir,         /* 39 = mkdir */
22257      do_unlink,        /* 40 = rmdir */
22258      do_dup,           /* 41 = dup */
22259      do_pipe,          /* 42 = pipe */
22260      no_sys,           /* 43 = times */
22261      no_sys,           /* 44 = (prof) */
22262      no_sys,           /* 45 = unused */
22263      do_set,           /* 46 = setgid */
22264      no_sys,           /* 47 = getgid */
22265      no_sys,           /* 48 = (signal) */
22266      no_sys,           /* 49 = unused */
22267      no_sys,           /* 50 = unused */
22268      no_sys,           /* 51 = (acct) */
22269      no_sys,           /* 52 = (phys) */
22270      no_sys,           /* 53 = (clock) */
22271      do_ioctl,          /* 54 = ioctl */
22272      do_fcntl,          /* 55 = fcntl */
22273      no_sys,           /* 56 = (mpx) */
22274      no_sys,           /* 57 = unused */
22275      no_sys,           /* 58 = unused */
22276      do_exec,           /* 59 = execve */
22277      do_umask,          /* 60 = umask */
22278      do_chroot,         /* 61 = chroot */
22279      do_setsid,         /* 62 = setsid */
22280      no_sys,           /* 63 = getpgrp */

22281      no_sys,           /* 64 = KSIG: signals originating in the kernel */
22282      do_unpause,        /* 65 = UNPAUSE */
22283      no_sys,           /* 66 = unused */
22284      do_revive,         /* 67 = REVIVE */
22285      no_sys,           /* 68 = TASK_REPLY */
22286      no_sys,           /* 69 = unused */
22287      no_sys,           /* 70 = unused */
22288      no_sys,           /* 71 = si */
22289      no_sys,           /* 72 = sigsuspend */
22290      no_sys,           /* 73 = sigpending */
22291      no_sys,           /* 74 = sigprocmask */
22292      no_sys,           /* 75 = sigreturn */
22293      no_sys,           /* 76 = reboot */
22294      do_reboot,         /* 77 = svrctl */

22295      no_sys,           /* 78 = unused */
22296      do_getsysinfo,     /* 79 = getsysinfo */
22297      no_sys,           /* 80 = unused */
22298      do_devctl,         /* 81 = devctl */
22299      no_sys,           /* 82 = fstatfs */
22300      do_fstatfs,        /* 83 = memalloc */
22301      no_sys,           /* 84 = memfree */
22302      do_select,         /* 85 = select */
22303      no_sys,           /* 86 = fchdir */
22304      do_fsync,          /* 87 = fsync */
22305      no_sys,           /* 88 = getpriority */
22306      no_sys,           /* 89 = setpriority */
22307      no_sys,           /* 90 = gettimeofday */
22308
22309
22310  };
22311 /* This should not fail with "array size is negative": */
22312 extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];
22313

```

```
=====
servers/fs/cache.c
=====

22400 /* The file system maintains a buffer cache to reduce the number of disk
22401 * accesses needed. Whenever a read or write to the disk is done, a check is
22402 * first made to see if the block is in the cache. This file manages the
22403 * cache.
22404 *
22405 * The entry points into this file are:
22406 *   get_block: request to fetch a block for reading or writing from cache
22407 *   put_block: return a block previously requested with get_block
22408 *   alloc_zone: allocate a new zone (to increase the length of a file)
22409 *   free_zone: release a zone (when a file is removed)
22410 *   rw_block: read or write a block from the disk itself
22411 *   invalidate: remove all the cache blocks on some device
22412 */
22413
22414 #include "fs.h"
22415 #include <minix/com.h>
22416 #include "buf.h"
22417 #include "file.h"
22418 #include "fproc.h"
22419 #include "super.h"
22420
22421 FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );
22422
22423 /*=====
22424 *           get_block
22425 *=====
22426 PUBLIC struct buf *get_block(dev, block, only_search)
22427 register dev_t dev;          /* on which device is the block? */
22428 register block_t block;      /* which block is wanted? */
22429 int only_search;            /* if NO_READ, don't read, else act normal */
22430 {
22431 /* Check to see if the requested block is in the block cache. If so, return
22432 * a pointer to it. If not, evict some other block and fetch it (unless
22433 * 'only_search' is 1). All the blocks in the cache that are not in use
22434 * are linked together in a chain, with 'front' pointing to the least recently
22435 * used block and 'rear' to the most recently used block. If 'only_search' is
22436 * 1, the block being requested will be overwritten in its entirety, so it is
22437 * only necessary to see if it is in the cache; if it is not, any free buffer
22438 * will do. It is not necessary to actually read the block in from disk.
22439 * If 'only_search' is PREFETCH, the block need not be read from the disk,
22440 * and the device is not to be marked on the block, so callers can tell if
22441 * the block returned is valid.
22442 * In addition to the LRU chain, there is also a hash chain to link together
22443 * blocks whose block numbers end with the same bit strings, for fast lookup.
22444 */
22445
22446     int b;
22447     register struct buf *bp, *prev_ptr;
22448
22449 /* Search the hash chain for (dev, block). Do_read() can use
22450 *   get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
22451 *   someone wants to read from a hole in a file, in which case this search
22452 *   is skipped
22453 */
22454 if (dev != NO_DEV) {
22455     b = (int) block & HASH_MASK;
22456     bp = buf_hash[b];
22457     while (bp != NIL_BUF) {
22458         if (bp->b_blocknr == block && bp->b_dev == dev) {
22459             /* Block needed has been found. */
22460             if (bp->b_count == 0) rm_lru(bp);
22461             bp->b_count++; /* record that block is in use */
22462 }
```

```

22463             return(bp);
22464         } else {
22465             /* This block is not the one sought. */
22466             bp = bp->b_hash; /* move to next block on hash chain */
22467         }
22468     }
22469 }
22470
22471 /* Desired block is not on available chain. Take oldest block ('front'). */
22472 if ((bp = front) == NIL_BUF) panic(__FILE__,"all buffers in use", NR_BUFS);
22473 rm_lru(bp);
22474
22475 /* Remove the block that was just taken from its hash chain. */
22476 b = (int) bp->b_blocknr & HASH_MASK;
22477 prev_ptr = buf_hash[b];
22478 if (prev_ptr == bp) {
22479     buf_hash[b] = bp->b_hash;
22480 } else {
22481     /* The block just taken is not on the front of its hash chain. */
22482     while (prev_ptr->b_hash != NIL_BUF)
22483         if (prev_ptr->b_hash == bp) {
22484             prev_ptr->b_hash = bp->b_hash; /* found it */
22485             break;
22486         } else {
22487             prev_ptr = prev_ptr->b_hash; /* keep looking */
22488         }
22489 }
22490
22491 /* If the block taken is dirty, make it clean by writing it to the disk.
22492 * Avoid hysteresis by flushing all other dirty blocks for the same device.
22493 */
22494 if (bp->b_dev != NO_DEV) {
22495     if (bp->b_dirt == DIRTY) flushall(bp->b_dev);
22496 }
22497
22498 /* Fill in block's parameters and add it to the hash chain where it goes. */
22499 bp->b_dev = dev;           /* fill in device number */
22500 bp->b_blocknr = block;    /* fill in block number */
22501 bp->b_count++;          /* record that block is being used */
22502 b = (int) bp->b_blocknr & HASH_MASK;
22503 bp->b_hash = buf_hash[b];
22504 buf_hash[b] = bp;         /* add to hash list */
22505
22506 /* Go get the requested block unless searching or prefetching. */
22507 if (dev != NO_DEV) {
22508     if (only_search == PREFETCH) bp->b_dev = NO_DEV;
22509     else
22510         if (only_search == NORMAL) {
22511             rw_block(bp, READING);
22512         }
22513 }
22514 return(bp);               /* return the newly acquired block */
22515 }

22516 /*=====
22517 *          put_block
22518 *=====
22519 PUBLIC void put_block(bp, block_type)
22520 register struct buf *bp;           /* pointer to the buffer to be released */
22521 int block_type;                  /* INODE_BLOCK, DIRECTORY_BLOCK, or whatever */
22522 {
22523 /* Return a block to the list of available blocks. Depending on 'block_type'
22524 * it may be put on the front or rear of the LRU chain. Blocks that are
22525 * expected to be needed again shortly (e.g., partially full data blocks)
22526 * go on the rear; blocks that are unlikely to be needed again shortly
22527 * (e.g., full data blocks) go on the front. Blocks whose loss can hurt
22528 * the integrity of the file system (e.g., inode blocks) are written to
22529 * disk immediately if they are dirty.

```

```

22531  */
22532  if (bp == NIL_BUF) return; /* it is easier to check here than in caller */
22533
22534  bp->b_count--;           /* there is one use fewer now */
22535  if (bp->b_count != 0) return; /* block is still in use */
22536
22537  bufs_in_use--;          /* one fewer block buffers in use */
22538
22539  /* Put this block back on the LRU chain. If the ONE_SHOT bit is set in
22540  * 'block_type', the block is not likely to be needed again shortly, so put
22541  * it on the front of the LRU chain where it will be the first one to be
22542  * taken when a free buffer is needed later.
22543  */
22544  if (bp->b_dev == DEV_RAM || block_type & ONE_SHOT) {
22545      /* Block probably won't be needed quickly. Put it on front of chain.
22546      * It will be the next block to be evicted from the cache.
22547      */
22548      bp->b_prev = NIL_BUF;
22549      bp->b_next = front;
22550      if (front == NIL_BUF)
22551          rear = bp; /* LRU chain was empty */
22552      else
22553          front->b_prev = bp;
22554      front = bp;
22555  } else {
22556      /* Block probably will be needed quickly. Put it on rear of chain.
22557      * It will not be evicted from the cache for a long time.
22558      */
22559      bp->b_prev = rear;
22560      bp->b_next = NIL_BUF;
22561      if (rear == NIL_BUF)
22562          front = bp;
22563      else
22564          rear->b_next = bp;
22565      rear = bp;
22566  }
22567
22568  /* Some blocks are so important (e.g., inodes, indirect blocks) that they
22569  * should be written to the disk immediately to avoid messing up the file
22570  * system in the event of a crash.
22571  */
22572  if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV) {
22573      rw_block(bp, WRITING);
22574  }
22575 }

22576 /*=====
22577 *          alloc_zone
22578 *=====
22579 PUBLIC zone_t alloc_zone(dev, z)
22580 dev_t dev;           /* device where zone wanted */
22581 zone_t z;            /* try to allocate new zone near this one */
22582 {
22583 /* Allocate a new zone on the indicated device and return its number. */
22584
22585     int major, minor;
22586     bit_t b, bit;
22587     struct super_block *sp;
22588
22589 /* Note that the routine alloc_bit() returns 1 for the lowest possible
22590 * zone, which corresponds to sp->s_firstdatazone. To convert a value
22591 * between the bit number, 'b', used by alloc_bit() and the zone number, 'z',
22592 * stored in the inode, use the formula:
22593 *      z = b + sp->s_firstdatazone - 1
22594 * Alloc_bit() never returns 0, since this is used for NO_BIT (failure).
22595 */
22596     sp = get_super(dev);

```

```

22598
22599 /* If z is 0, skip initial part of the map known to be fully in use. */
22600 if (z == sp->s_firstdatazone) {
22601     bit = sp->s_zsearch;
22602 } else {
22603     bit = (bit_t) z - (sp->s_firstdatazone - 1);
22604 }
22605 b = alloc_bit(sp, ZMAP, bit);
22606 if (b == NO_BIT) {
22607     err_code = ENOSPC;
22608     major = (int) (sp->s_dev >> MAJOR) & BYTE;
22609     minor = (int) (sp->s_dev >> MINOR) & BYTE;
22610     printf("No space on %sdevice %d/%d\n",
22611         sp->s_dev == root_dev ? "root " : "", major, minor);
22612     return(NO_ZONE);
22613 }
22614 if (z == sp->s_firstdatazone) sp->s_zsearch = b; /* for next time */
22615 return(sp->s_firstdatazone - 1 + (zone_t) b);
22616 }

22618 /*=====
22619 *          free_zone
22620 *=====
22621 PUBLIC void free_zone(dev, numb)
22622 dev_t dev;                                /* device where zone located */
22623 zone_t numb;                             /* zone to be returned */
22624 {
22625 /* Return a zone. */
22626
22627 register struct super_block *sp;
22628 bit_t bit;
22629
22630 /* Locate the appropriate super_block and return bit. */
22631 sp = get_super(dev);
22632 if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
22633 bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
22634 free_bit(sp, ZMAP, bit);
22635 if (bit < sp->s_zsearch) sp->s_zsearch = bit;
22636 }

22638 /*=====
22639 *          rw_block
22640 *=====
22641 PUBLIC void rw_block(bp, rw_flag)
22642 register struct buf *bp; /* buffer pointer */
22643 int rw_flag;           /* READING or WRITING */
22644 {
22645 /* Read or write a disk block. This is the only routine in which actual disk
22646 * I/O is invoked. If an error occurs, a message is printed here, but the error
22647 * is not reported to the caller. If the error occurred while purging a block
22648 * from the cache, it is not clear what the caller could do about it anyway.
22649 */
22650
22651     int r, op;
22652     off_t pos;
22653     dev_t dev;
22654     int block_size;
22655
22656     block_size = get_block_size(bp->b_dev);
22657
22658     if ( (dev = bp->b_dev) != NO_DEV) {
22659         pos = (off_t) bp->b_blocknr * block_size;
22660         op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
22661         r = dev_io(op, dev, FS_PROC_NR, bp->b_data, pos, block_size, 0);
22662         if (r != block_size) {
22663             if (r >= 0) r = END_OF_FILE;
22664             if (r != END_OF_FILE)

```

```

22665         printf("Unrecoverable disk error on device %d/%d, block %ld\n",
22666             (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
22667             bp->b_dev = NO_DEV; /* invalidate block */
22668
22669             /* Report read errors to interested parties. */
2270             if (rw_flag == READING) rdwt_err = r;
2271         }
2272     }
2273
2274     bp->b_dirt = CLEAN;
2275 }

22767 /*=====
22768 *           invalidate
22769 *=====
2280 PUBLIC void invalidate(device)
2281 dev_t device; /* device whose blocks are to be purged */
2282 {
2283     /* Remove all the blocks belonging to some device from the cache. */
2284
2285     register struct buf *bp;
2286
2287     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
2288         if (bp->b_dev == device) bp->b_dev = NO_DEV;
2289 }

22901 /*=====
22902 *           flushall
22903 *=====
2294 PUBLIC void flushall(dev)
2295 dev_t dev; /* device to flush */
2296 {
2297     /* Flush all dirty blocks for one device. */
2298
2299     register struct buf *bp;
2300     static struct buf *dirty[NR_BUFS]; /* static so it isn't on stack */
2301     int ndirty;
2302
2303     for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
2304         if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
2305     rw_scattered(dev, dirty, ndirty, WRITING);
2306 }

23078 /*=====
23079 *           rw_scattered
23080 *=====
2311 PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
2312 dev_t dev; /* major-minor device number */
2313 struct buf **bufq; /* pointer to array of buffers */
2314 int bufqsize; /* number of buffers */
2315 int rw_flag; /* READING or WRITING */
2316 {
2317     /* Read or write scattered data from a device. */
2318
2319     register struct buf *bp;
2320     int gap;
2321     register int i;
2322     register iovec_t *iop;
2323     static iovec_t iovec[NR_IOREQS]; /* static so it isn't on stack */
2324     int j, r;
2325     int block_size;
2326
2327     block_size = get_block_size(dev);
2328
2329     /* (Shell) sort buffers on b_blocknr. */
2330     gap = 1;
2331     do

```

```

22732      gap = 3 * gap + 1;
22733      while (gap <= bufqsize);
22734      while (gap != 1) {
22735          gap /= 3;
22736          for (j = gap; j < bufqsize; j++) {
22737              for (i = j - gap;
22738                  i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
22739                  i -= gap) {
22740                  bp = bufq[i];
22741                  bufq[i] = bufq[i + gap];
22742                  bufq[i + gap] = bp;
22743              }
22744          }
22745      }
22746
22747      /* Set up I/O vector and do I/O. The result of dev_io is OK if everything
22748      * went fine, otherwise the error code for the first failed transfer.
22749      */
22750      while (bufqsize > 0) {
22751          for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
22752              bp = bufq[j];
22753              if (bp->b_blocknr != bufq[0]->b_blocknr + j) break;
22754              iop->iov_addr = (vir_bytes) bp->b_data;
22755              iop->iov_size = block_size;
22756          }
22757          r = dev_io(rw_flag == WRITING ? DEV_SCATTER : DEV_GATHER,
22758                      dev, FS_PROC_NR, iovec,
22759                      (off_t) bufq[0]->b_blocknr * block_size, j, 0);
22760
22761          /* Harvest the results. Dev_io reports the first error it may have
22762          * encountered, but we only care if it's the first block that failed.
22763          */
22764          for (i = 0, iop = iovec; i < j; i++, iop++) {
22765              bp = bufq[i];
22766              if (iop->iov_size != 0) {
22767                  /* Transfer failed. An error? Do we care? */
22768                  if (r != OK && i == 0) {
22769                      printf(
22770                          "fs: I/O error on device %d/%d, block %lu\n",
22771                          (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE,
22772                          bp->b_blocknr);
22773                      bp->b_dev = NO_DEV; /* invalidate block */
22774                  }
22775                  break;
22776              }
22777              if (rw_flag == READING) {
22778                  bp->b_dev = dev; /* validate block */
22779                  put_block(bp, PARTIAL_DATA_BLOCK);
22780              } else {
22781                  bp->b_dirt = CLEAN;
22782              }
22783          }
22784          bufq += i;
22785          bufqsize -= i;
22786          if (rw_flag == READING) {
22787              /* Don't bother reading more than the device is willing to
22788              * give at this time. Don't forget to release those extras.
22789              */
22790              while (bufqsize > 0) {
22791                  put_block(*bufq++, PARTIAL_DATA_BLOCK);
22792                  bufqsize--;
22793              }
22794          }
22795          if (rw_flag == WRITING && i == 0) {
22796              /* We're not making progress, this means we might keep
22797              * looping. Buffers remain dirty if un-written. Buffers are
22798              * lost if invalidate()d or LRU-removed while dirty. This

```

```
22799             * is better than keeping unwritable blocks around forever..  
22800             */  
22801         break;  
22802     }  
22803 }  
22804 }  
  
22806 /*======  
22807 *                         rm_lru  
22808 *======  
22809 PRIVATE void rm_lru(bp)  
22810 struct buf *bp;  
22811 {  
22812 /* Remove a block from its LRU chain. */  
22813     struct buf *next_ptr, *prev_ptr;  
22814  
22815     bufs_in_use++;  
22816     next_ptr = bp->b_next;          /* successor on LRU chain */  
22817     prev_ptr = bp->b_prev;          /* predecessor on LRU chain */  
22818     if (prev_ptr != NIL_BUF)  
22819         prev_ptr->b_next = next_ptr;  
22820     else  
22821         front = next_ptr;          /* this block was at front of chain */  
22822  
22823     if (next_ptr != NIL_BUF)  
22824         next_ptr->b_prev = prev_ptr;  
22825     else  
22826         rear = prev_ptr;          /* this block was at rear of chain */  
22827 }
```

```

22930 /*=====
22931 *          *
22932 *=====*/
22933 PUBLIC struct inode *get_inode(dev, num)
22934 dev_t dev;           /* device on which inode resides */
22935 int num;             /* inode number (ANSI: may not be unshort) */
22936 {
22937 /* Find a slot in the inode table, load the specified inode into it, and
22938 * return a pointer to the slot. If 'dev' == NO_DEV, just return a free slot.
22939 */
22940
22941     register struct inode *rip, *xp;
22942
22943     /* Search the inode table both for (dev, num) and a free slot. */
22944     xp = NIL_INODE;
22945     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
22946         if (rip->i_count > 0) { /* only check used slots for (dev, num) */
22947             if (rip->i_dev == dev && rip->i_num == num) {
22948                 /* This is the inode that we are looking for. */
22949                 rip->i_count++;
22950                 return(rip); /* (dev, num) found */
22951             }
22952         } else {
22953             xp = rip; /* remember this free slot for later */
22954         }
22955     }
22956
22957     /* Inode we want is not currently in use. Did we find a free slot? */
22958     if (xp == NIL_INODE) { /* inode table completely full */
22959         err_code = ENFILE;
22960         return(NIL_INODE);
22961     }
22962
22963     /* A free inode slot has been located. Load the inode into it. */
22964     xp->i_dev = dev;
22965     xp->i_num = num;
22966     xp->i_count = 1;
22967     if (dev != NO_DEV) rw_inode(xp, READING); /* get inode from disk */
22968     xp->i_update = 0; /* all the times are initially up-to-date */
22969
22970     return(xp);
22971 }

22973 /*=====
22974 *          *
22975 *=====*/
22976 PUBLIC void put_inode(rip)
22977 register struct inode *rip; /* pointer to inode to be released */
22978 {
22979     /* The caller is no longer using this inode. If no one else is using it either
22980      * write it back to the disk immediately. If it has no links, truncate it and
22981      * return it to the pool of available inodes.
22982 */
22983
22984     if (rip == NIL_INODE) return; /* checking here is easier than in caller */
22985     if (--rip->i_count == 0) { /* i_count == 0 means no one is using it now */
22986         if (rip->i_nlinks == 0) {
22987             /* i_links == 0 means free the inode. */
22988             truncate(rip); /* return all the disk blocks */
22989             rip->i_mode = I_NOT_ALLOC; /* clear I_TYPE field */
22990             rip->i_dirt = DIRTY;
22991             free_inode(rip->i_dev, rip->i_num);
22992         } else {
22993             if (rip->i_pipe == I_PIPE) truncate(rip);
22994         }
22995         rip->i_pipe = NO_PIPE; /* should always be cleared */
22996         if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);

```

```

22997 }
22998 }

23000 /*=====
23001 *          alloc_inode
23002 *=====
23003 PUBLIC struct inode *alloc_inode(dev_t dev, mode_t bits)
23004 {
23005 /* Allocate a free inode on 'dev', and return a pointer to it. */
23006
23007     register struct inode *rip;
23008     register struct super_block *sp;
23009     int major, minor, inumb;
23010     bit_t b;
23011
23012     sp = get_super(dev); /* get pointer to super_block */
23013     if (sp->s_rd_only) { /* can't allocate an inode on a read only device. */
23014         err_code = EROFS;
23015         return(NIL_INODE);
23016     }
23017
23018 /* Acquire an inode from the bit map. */
23019     b = alloc_bit(sp, IMAP, sp->s_isearch);
23020     if (b == NO_BIT) {
23021         err_code = ENFILE;
23022         major = (int) (sp->s_dev >> MAJOR) & BYTE;
23023         minor = (int) (sp->s_dev >> MINOR) & BYTE;
23024         printf("Out of i-nodes on %sdevice %d/%d\n",
23025             sp->s_dev == root_dev ? "root " : "", major, minor);
23026         return(NIL_INODE);
23027     }
23028     sp->s_isearch = b;           /* next time start here */
23029     inumb = (int) b;            /* be careful not to pass unshort as param */
23030
23031 /* Try to acquire a slot in the inode table. */
23032     if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
23033         /* No inode table slots available. Free the inode just allocated. */
23034         free_bit(sp, IMAP, b);
23035     } else {
23036         /* An inode slot is available. Put the inode just allocated into it. */
23037         rip->i_mode = bits;           /* set up RWX bits */
23038         rip->i_nlinks = 0;           /* initial no links */
23039         rip->i_uid = fp->fp_effuid; /* file's uid is owner's */
23040         rip->i_gid = fp->fp_effgid; /* ditto group id */
23041         rip->i_dev = dev;           /* mark which device it is on */
23042         rip->i_ndzones = sp->s_ndzones; /* number of direct zones */
23043         rip->i_nindirs = sp->s_nindirs; /* number of indirect zones per blk*/
23044         rip->i_sp = sp;             /* pointer to super block */
23045
23046         /* Fields not cleared already are cleared in wipe_inode(). They have
23047         * been put there because truncate() needs to clear the same fields if
23048         * the file happens to be open while being truncated. It saves space
23049         * not to repeat the code twice.
23050         */
23051         wipe_inode(rip);
23052     }
23053
23054     return(rip);
23055 }

23056 /*=====
23057 *          wipe_inode
23058 *=====
23059 PUBLIC void wipe_inode(rip)
23060     register struct inode *rip; /* the inode to be erased */
23061
23062     /*
23063      * Erase some fields in the inode. This function is called from alloc_inode()

```

```

23064 * when a new inode is to be allocated, and from truncate(), when an existing
23065 * inode is to be truncated.
23066 */
23067
23068 register int i;
23069
23070 rip->i_size = 0;
23071 rip->i_update = ATIME | CTIME | MTIME; /* update all times later */
23072 rip->i_dirty = DIRTY;
23073 for (i = 0; i < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
23074 }

23076 /*=====
23077 *          free_inode
23078 =====*/
23079 PUBLIC void free_inode(dev, inumb)
23080 dev_t dev;           /* on which device is the inode */
23081 ino_t inumb;         /* number of inode to be freed */
23082 {
23083 /* Return an inode to the pool of unallocated inodes. */
23084
23085 register struct super_block *sp;
23086 bit_t b;
23087
23088 /* Locate the appropriate super_block. */
23089 sp = get_super(dev);
23090 if (inumb <= 0 || inumb > sp->s_ninodes) return;
23091 b = inumb;
23092 free_bit(sp, IMAP, b);
23093 if (b < sp->s_isearch) sp->s_isearch = b;
23094 }

23096 /*=====
23097 *          update_times
23098 =====*/
23099 PUBLIC void update_times(rip)
23100 register struct inode *rip; /* pointer to inode to be read/written */
23101 {
23102 /* Various system calls are required by the standard to update atime,
23103 * or mtime. Since updating a time requires sending a message to the clock
23104 * task--an expensive business--the times are marked for update by setting
23105 * bits in i_update. When a stat, fstat, or sync is done, or an inode is
23106 * released, update_times() may be called to actually fill in the times.
23107 */
23108
23109     time_t cur_time;
23110     struct super_block *sp;
23111
23112     sp = rip->i_sp;           /* get pointer to super block. */
23113     if (sp->s_rd_only) return; /* no updates for read-only file systems */
23114
23115     cur_time = clock_time();
23116     if (rip->i_update & ATIME) rip->i_atime = cur_time;
23117     if (rip->i_update & CTIME) rip->i_ctime = cur_time;
23118     if (rip->i_update & MTIME) rip->i_mtime = cur_time;
23119     rip->i_update = 0;        /* they are all up-to-date now */
23120 }

23122 /*=====
23123 *          rw_inode
23124 =====*/
23125 PUBLIC void rw_inode(rip, rw_flag)
23126 register struct inode *rip; /* pointer to inode to be read/written */
23127 int rw_flag;              /* READING or WRITING */
23128 {
23129 /* An entry in the inode table is to be copied to or from the disk. */
23130

```

```

23131 register struct buf *bp;
23132 register struct super_block *sp;
23133 d1_inode *dip;
23134 d2_inode *dip2;
23135 block_t b, offset;
23136
23137 /* Get the block where the inode resides. */
23138 sp = get_super(rip->i_dev); /* get pointer to super block */
23139 rip->i_sp = sp; /* inode must contain super block pointer */
23140 offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
23141 b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
23142 bp = get_block(rip->i_dev, b, NORMAL);
23143 dip = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
23144 dip2 = bp->b_v2_ino + (rip->i_num - 1) %
23145 V2_INODES_PER_BLOCK(sp->s_block_size);
23146
23147 /* Do the read or write. */
23148 if (rw_flag == WRITING) {
23149     if (rip->i_update) update_times(rip); /* times need updating */
23150     if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
23151 }
23152
23153 /* Copy the inode from the disk block to the in-core table or vice versa.
23154 * If the fourth parameter below is FALSE, the bytes are swapped.
23155 */
23156 if (sp->s_version == V1)
23157     old_icopy(rip, dip, rw_flag, sp->s_native);
23158 else
23159     new_icopy(rip, dip2, rw_flag, sp->s_native);
23160
23161 put_block(bp, INODE_BLOCK);
23162 rip->i_dirt = CLEAN;
23163 }

23165 /*=====
23166 *          old_icopy
23167 =====*/
23168 PRIVATE void old_icopy(rip, dip, direction, norm)
23169 register struct inode *rip; /* pointer to the in-core inode struct */
23170 register d1_inode *dip; /* pointer to the d1_inode inode struct */
23171 int direction; /* READING (from disk) or WRITING (to disk) */
23172 int norm; /* TRUE = do not swap bytes; FALSE = swap */
23173
23174 {
23175 /* The V1.x IBM disk, the V1.x 68000 disk, and the V2 disk (same for IBM and
23176 * 68000) all have different inode layouts. When an inode is read or written
23177 * this routine handles the conversions so that the information in the inode
23178 * table is independent of the disk structure from which the inode came.
23179 * The old_icopy routine copies to and from V1 disks.
23180 */
23181
23182     int i;
23183
23184     if (direction == READING) {
23185         /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
23186         rip->i_mode = conv2(norm, (int) dip->d1_mode);
23187         rip->i_uid = conv2(norm, (int) dip->d1_uid );
23188         rip->i_size = conv4(norm, dip->d1_size);
23189         rip->i_mtime = conv4(norm, dip->d1_mtime);
23190         rip->i_atime = rip->i_mtime;
23191         rip->i_ctime = rip->i_mtime;
23192         rip->i_nlinks = dip->d1_nlinks; /* 1 char */
23193         rip->i_gid = dip->d1_gid; /* 1 char */
23194         rip->i_ndzones = V1_NR_DZONES;
23195         rip->i_nindirs = V1_INDIRECTS;
23196         for (i = 0; i < V1_NR_TZONES; i++)
23197             rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);

```

```

23198    } else {
23199        /* Copying V1.x inode to disk from the in-core table. */
23200        dip->dl_mode    = conv2(norm, (int) rip->i_mode);
23201        dip->dl_uid     = conv2(norm, (int) rip->i_uid );
23202        dip->dl_size    = conv4(norm,          rip->i_size);
23203        dip->dl_mtime   = conv4(norm,          rip->i_mtime);
23204        dip->dl_nlinks  = rip->i_nlinks;           /* 1 char */
23205        dip->dl_gid     = rip->i_gid;             /* 1 char */
23206        for (i = 0; i < V1_NR_TZONES; i++)
23207            dip->dl_zone[i] = conv2(norm, (int) rip->i_zone[i]);
23208    }
23209 }

23211 /*=====
23212 *                         new_icopy
23213 *=====
23214 PRIVATE void new_icopy(rip, dip, direction, norm)
23215 register struct inode *rip; /* pointer to the in-core inode struct */
23216 register d2_inode *dip; /* pointer to the d2_inode struct */
23217 int direction;           /* READING (from disk) or WRITING (to disk) */
23218 int norm;                /* TRUE = do not swap bytes; FALSE = swap */
23219
23220 {
23221 /* Same as old_icopy, but to/from V2 disk layout. */
23222
23223     int i;
23224
23225     if (direction == READING) {
23226         /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
23227         rip->i_mode    = conv2(norm,dip->d2_mode);
23228         rip->i_uid     = conv2(norm,dip->d2_uid);
23229         rip->i_nlinks  = conv2(norm,dip->d2_nlinks);
23230         rip->i_gid     = conv2(norm,dip->d2_gid);
23231         rip->i_size    = conv4(norm,dip->d2_size);
23232         rip->i_atime   = conv4(norm,dip->d2_atime);
23233         rip->i_ctime   = conv4(norm,dip->d2_ctime);
23234         rip->i_mtime   = conv4(norm,dip->d2_mtime);
23235         rip->i_ndzones = V2_NR_DZONES;
23236         rip->i_nindirs = V2_INDIRECTS(rip->i_sp->s_block_size);
23237         for (i = 0; i < V2_NR_TZONES; i++)
23238             rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
23239     } else {
23240         /* Copying V2.x inode to disk from the in-core table. */
23241         dip->d2_mode    = conv2(norm,rip->i_mode);
23242         dip->d2_uid     = conv2(norm,rip->i_uid);
23243         dip->d2_nlinks  = conv2(norm,rip->i_nlinks);
23244         dip->d2_gid     = conv2(norm,rip->i_gid);
23245         dip->d2_size    = conv4(norm,rip->i_size);
23246         dip->d2_atime   = conv4(norm,rip->i_atime);
23247         dip->d2_ctime   = conv4(norm,rip->i_ctime);
23248         dip->d2_mtime   = conv4(norm,rip->i_mtime);
23249         for (i = 0; i < V2_NR_TZONES; i++)
23250             dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
23251     }
23252 }

23254 /*=====
23255 *                         dup_inode
23256 *=====
23257 PUBLIC void dup_inode(ip)
23258 struct inode *ip; /* The inode to be duplicated. */
23259 {
23260 /* This routine is a simplified form of get_inode() for the case where
23261 * the inode pointer is already known.
23262 */
23263     ip->i_count++;

```

23265 }

```
*****+
servers/fs/super.c
*****+
```

23300 /\* This file manages the super block table and the related data structures,  
23301 \* namely, the bit maps that keep track of which zones and which inodes are  
23302 \* allocated and which are free. When a new inode or zone is needed, the  
23303 \* appropriate bit map is searched for a free entry.  
23304 \*/  
23305 \* The entry points into this file are  
23306 \* alloc\_bit: somebody wants to allocate a zone or inode; find one  
23307 \* free\_bit: indicate that a zone or inode is available for allocation  
23308 \* get\_super: search the 'superblock' table for a device  
23309 \* mounted: tells if file inode is on mounted (or ROOT) file system  
23310 \* read\_super: read a superblock  
23311 \*/  
23312  
23313 #include "fs.h"  
23314 #include <string.h>  
23315 #include <minix/com.h>  
23316 #include "buf.h"  
23317 #include "inode.h"  
23318 #include "super.h"  
23319 #include "const.h"  
23320  
23321 /\*=====\*  
23322 \* alloc\_bit \*  
23323 \*=====\*/  
23324 PUBLIC bit\_t alloc\_bit(sp, map, origin)  
23325 struct super\_block \*sp; /\* the filesystem to allocate from \*/  
23326 int map; /\* IMAP (inode map) or ZMAP (zone map) \*/  
23327 bit\_t origin; /\* number of bit to start searching at \*/  
23328 {  
23329 /\* Allocate a bit from a bit map and return its bit number. \*/  
23330  
23331 block\_t start\_block; /\* first bit block \*/  
23332 bit\_t map\_bits; /\* how many bits are there in the bit map? \*/  
23333 unsigned bit\_blocks; /\* how many blocks are there in the bit map? \*/  
23334 unsigned block, word, bcount;  
23335 struct buf \*bp;  
23336 bitchunk\_t \*wptr, \*wlim, k;  
23337 bit\_t i, b;  
23338  
23339 if (sp->s\_rd\_only)  
23340 panic(FILE, "can't allocate bit on read-only filesystem.", NO\_NUM);  
23341  
23342 if (map == IMAP) {  
23343 start\_block = START\_BLOCK;  
23344 map\_bits = sp->s\_ninodes + 1;  
23345 bit\_blocks = sp->s\_imap\_blocks;  
23346 } else {  
23347 start\_block = START\_BLOCK + sp->s\_imap\_blocks;  
23348 map\_bits = sp->s\_zones - (sp->s\_firstdatazone - 1);  
23349 bit\_blocks = sp->s\_zmap\_blocks;  
23350 }  
23351  
23352 /\* Figure out where to start the bit search (depends on 'origin'). \*/  
23353 if (origin >= map\_bits) origin = 0; /\* for robustness \*/  
23354  
23355 /\* Locate the starting place. \*/  
23356 block = origin / FS\_BITS\_PER\_BLOCK(sp->s\_block\_size);

```

23357     word = (origin % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;
23358
23359     /* Iterate over all blocks plus one, because we start in the middle. */
23360     bcount = bit_blocks + 1;
23361     do {
23362         bp = get_block(sp->s_dev, start_block + block, NORMAL);
23363         wlim = &bp->b_bitmap[FS_BITMAP_CHUNKS(sp->s_block_size)];
23364
23365         /* Iterate over the words in block. */
23366         for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {
23367
23368             /* Does this word contain a free bit? */
23369             if (*wptr == (bitchunk_t) ~0) continue;
23370
23371             /* Find and allocate the free bit. */
23372             k = conv2(sp->s_native, (int) *wptr);
23373             for (i = 0; (k & (1 << i)) != 0; ++i) {}
23374
23375             /* Bit number from the start of the bit map. */
23376             b = ((bit_t) block * FS_BITS_PER_BLOCK(sp->s_block_size))
23377                 + (wptr - &bp->b_bitmap[0]) * FS_BITCHUNK_BITS
23378                 + i;
23379
23380             /* Don't allocate bits beyond the end of the map. */
23381             if (b >= map_bits) break;
23382
23383             /* Allocate and return bit number. */
23384             k |= 1 << i;
23385             *wptr = conv2(sp->s_native, (int) k);
23386             bp->b_dirty = DIRTY;
23387             put_block(bp, MAP_BLOCK);
23388             return(b);
23389         }
23390         put_block(bp, MAP_BLOCK);
23391         if (++block >= bit_blocks) block = 0; /* last block, wrap around */
23392         word = 0;
23393     } while (--bcount > 0);
23394     return(NO_BIT); /* no bit could be allocated */
23395 }

23397 /*=====
23398 *          free_bit
23399 *=====
23400 PUBLIC void free_bit(sp, map, bit_returned)
23401 struct super_block *sp;           /* the filesystem to operate on */
23402 int map;                         /* IMAP (inode map) or ZMAP (zone map) */
23403 bit_t bit_returned;             /* number of bit to insert into the map */
23404 {
23405 /* Return a zone or inode by turning off its bitmap bit. */
23406
23407     unsigned block, word, bit;
23408     struct buf *bp;
23409     bitchunk_t k, mask;
23410     block_t start_block;
23411
23412     if (sp->s_rd_only)
23413         panic(__FILE__, "can't free bit on read-only filesys.", NO_NUM);
23414
23415     if (map == IMAP) {
23416         start_block = START_BLOCK;
23417     } else {
23418         start_block = START_BLOCK + sp->s_imap_blocks;
23419     }
23420     block = bit_returned / FS_BITS_PER_BLOCK(sp->s_block_size);
23421     word = (bit_returned % FS_BITS_PER_BLOCK(sp->s_block_size))
23422         / FS_BITCHUNK_BITS;
23423

```

```

23424     bit = bit_returned % FS_BITCHUNK_BITS;
23425     mask = 1 << bit;
23426
23427     bp = get_block(sp->s_dev, start_block + block, NORMAL);
23428
23429     k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
23430     if (!(k & mask)) {
23431         panic(FILE__, map == IMAP ? "tried to free unused inode" :
23432             "tried to free unused block", NO_NUM);
23433     }
23434
23435     k &= ~mask;
23436     bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
23437     bp->b_dirt = DIRTY;
23438
23439     put_block(bp, MAP_BLOCK);
23440 }

23442 /*=====
23443 *          *
23444 *          get_super
23445 =====*/
23446 PUBLIC struct super_block *get_super(dev)
23447 dev_t dev;           /* device number whose super_block is sought */
23448 {
23449 /* Search the superblock table for this device. It is supposed to be there. */
23450     register struct super_block *sp;
23451
23452     if (dev == NO_DEV)
23453         panic(FILE__, "request for super_block of NO_DEV", NO_NUM);
23454
23455     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
23456         if (sp->s_dev == dev) return(sp);
23457
23458 /* Search failed. Something wrong. */
23459 panic(FILE__, "can't find superblock for device (in decimal)", (int) dev);
23460
23461     return(NIL_SUPER);      /* to keep the compiler and lint quiet */
23462 }

23464 /*=====
23465 *          *
23466 *          get_block_size
23467 =====*/
23468 PUBLIC int get_block_size(dev_t dev)
23469 {
23470 /* Search the superblock table for this device. */
23471     register struct super_block *sp;
23472
23473     if (dev == NO_DEV)
23474         panic(FILE__, "request for block size of NO_DEV", NO_NUM);
23475
23476     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
23477         if (sp->s_dev == dev) {
23478             return(sp->s_block_size);
23479         }
23480     }
23481
23482 /* no mounted filesystem? use this block size then. */
23483     return MIN_BLOCK_SIZE;
23484 }

23486 /*=====
23487 *          *
23488 *          mounted
23489 =====*/
23490 PUBLIC int mounted(rip);    /* pointer to inode */

```

```

23491 {
23492 /* Report on whether the given inode is on a mounted (or ROOT) file system. */
23493
23494     register struct super_block *sp;
23495     register dev_t dev;
23496
23497     dev = (dev_t) rip->i_zone[0];
23498     if (dev == root_dev) return(TRUE); /* inode is on root file system */
23499
23500     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
23501         if (sp->s_dev == dev) return(TRUE);
23502
23503     return(FALSE);
23504 }

23506 /*=====
23507 *          read_super
23508 *=====
23509 PUBLIC int read_super(sp)
23510 register struct super_block *sp; /* pointer to a superblock */
23511 {
23512 /* Read a superblock. */
23513     dev_t dev;
23514     int magic;
23515     int version, native, r;
23516     static char sdbuf[MIN_BLOCK_SIZE];
23517
23518     dev = sp->s_dev;           /* save device (will be overwritten by copy) */
23519     if (dev == NO_DEV)
23520         panic(FILE, "request for super_block of NO_DEV", NO_NUM);
23521     r = dev_io(DEV_READ, dev, FS_PROC_NR,
23522                 sdbuf, SUPER_BLOCK_BYTES, MIN_BLOCK_SIZE, 0);
23523     if (r != MIN_BLOCK_SIZE) {
23524         return EINVAL;
23525     }
23526     memcpy(sp, sdbuf, sizeof(*sp));
23527     sp->s_dev = NO_DEV;        /* restore later */
23528     magic = sp->s_magic;      /* determines file system type */
23529
23530 /* Get file system version and type. */
23531 if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
23532     version = V1;
23533     native = (magic == SUPER_MAGIC);
23534 } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
23535     version = V2;
23536     native = (magic == SUPER_V2);
23537 } else if (magic == SUPER_V3) {
23538     version = V3;
23539     native = 1;
23540 } else {
23541     return(EINVAL);
23542 }
23543
23544 /* If the super block has the wrong byte order, swap the fields; the magic
23545 * number doesn't need conversion. */
23546 sp->s_ninodes = conv4(native, sp->s_ninodes);
23547 sp->s_nzones = conv2(native, (int) sp->s_nzones);
23548 sp->s_imap_blocks = conv2(native, (int) sp->s_imap_blocks);
23549 sp->s_zmap_blocks = conv2(native, (int) sp->s_zmap_blocks);
23550 sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
23551 sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
23552 sp->s_max_size = conv4(native, sp->s_max_size);
23553 sp->s_zones = conv4(native, sp->s_zones);
23554
23555 /* In V1, the device size was kept in a short, s_nzones, which limited
23556 * devices to 32K zones. For V2, it was decided to keep the size as a
23557 * long. However, just changing s_nzones to a long would not work, since
23558 * then the position of s_magic in the super block would not be the same

```

```

23559 * in V1 and V2 file systems, and there would be no way to tell whether
23560 * a newly mounted file system was V1 or V2. The solution was to introduce
23561 * a new variable, s_zones, and copy the size there.
23562 *
23563 * Calculate some other numbers that depend on the version here too, to
23564 * hide some of the differences.
23565 */
23566 if (version == V1) {
23567     sp->s_block_size = STATIC_BLOCK_SIZE;
23568     sp->s_zones = sp->s_nzones; /* only V1 needs this copy */
23569     sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
23570     sp->s_ndzones = V1_NR_DZONES;
23571     sp->s_nindirs = V1_INDIRECTS;
23572 } else {
23573     if (version == V2)
23574         sp->s_block_size = STATIC_BLOCK_SIZE;
23575     if (sp->s_block_size < MIN_BLOCK_SIZE)
23576         return EINVAL;
23577     sp->s_inodes_per_block = V2_INODES_PER_BLOCK(sp->s_block_size);
23578     sp->s_ndzones = V2_NR_DZONES;
23579     sp->s_nindirs = V2_INDIRECTS(sp->s_block_size);
23580 }
23581
23582 if (sp->s_block_size < MIN_BLOCK_SIZE) {
23583     return EINVAL;
23584 }
23585 if (sp->s_block_size > MAX_BLOCK_SIZE) {
23586     printf("Filesystem block size is %d kB; maximum filesystem\n"
23587           "block size is %d kB. This limit can be increased by recompiling.\n",
23588           sp->s_block_size/1024, MAX_BLOCK_SIZE/1024);
23589     return EINVAL;
23590 }
23591 if ((sp->s_block_size % 512) != 0) {
23592     return EINVAL;
23593 }
23594 if (SUPER_SIZE > sp->s_block_size) {
23595     return EINVAL;
23596 }
23597 if ((sp->s_block_size % V2_INODE_SIZE) != 0 ||
23598      (sp->s_block_size % V1_INODE_SIZE) != 0) {
23599     return EINVAL;
23600 }
23601
23602 sp->s_isearch = 0; /* inode searches initially start at 0 */
23603 sp->s_zsearch = 0; /* zone searches initially start at 0 */
23604 sp->s_version = version;
23605 sp->s_native = native;
23606
23607 /* Make a few basic checks to see if super block looks reasonable. */
23608 if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
23609             || sp->s_ninodes < 1 || sp->s_zones < 1
23610             || (unsigned) sp->s_log_zone_size > 4) {
23611     printf("not enough imap or zone map blocks, \n");
23612     printf("or not enough inodes, or not enough zones, "
23613           "or zone size too large\n");
23614     return(EINVAL);
23615 }
23616 sp->s_dev = dev; /* restore device number */
23617 return(OK);
23618 }

+++++++++++++++++++++++++++++++++
servers/fs/filedes.c
+++++++++++++++++++++++++++++++++
23700 /* This file contains the procedures that manipulate file descriptors.

```

```

23701  /*
23702   * The entry points into this file are
23703   *   get_fd:    look for free file descriptor and free filp slots
23704   *   get_filp:  look up the filp entry for a given file descriptor
23705   *   find_filp: find a filp slot that points to a given inode
23706   */
23707
23708 #include "fs.h"
23709 #include "file.h"
23710 #include "fproc.h"
23711 #include "inode.h"
23712
23713 /*=====
23714   *           get_fd
23715   *=====
23716 PUBLIC int get_fd(int start, mode_t bits, int *k, struct filp **fpt)
23717 {
23718 /* Look for a free file descriptor and a free filp slot.  Fill in the mode word
23719 * in the latter, but don't claim either one yet, since the open() or creat()
23720 * may yet fail.
23721 */
23722
23723 register struct filp *f;
23724 register int i;
23725
23726 *k = -1;           /* we need a way to tell if file desc found */
23727
23728 /* Search the fproc fp_filp table for a free file descriptor. */
23729 for (i = start; i < OPEN_MAX; i++) {
23730     if (fp->fp_filp[i] == NIL_FILP) {
23731         /* A file descriptor has been located. */
23732         *k = i;
23733         break;
23734     }
23735 }
23736
23737 /* Check to see if a file descriptor has been found. */
23738 if (*k < 0) return(EMFILE); /* this is why we initialized k to -1 */
23739
23740 /* Now that a file descriptor has been found, look for a free filp slot. */
23741 for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
23742     if (f->filp_count == 0) {
23743         f->filp_mode = bits;
23744         f->filp_pos = 0L;
23745         f->filp_selectors = 0;
23746         f->filp_select_ops = 0;
23747         f->filp_pipe_select_ops = 0;
23748         f->filp_flags = 0;
23749         *fpt = f;
23750         return(OK);
23751     }
23752 }
23753
23754 /* If control passes here, the filp table must be full. Report that back. */
23755 return(ENFILE);
23756 }

23757 /*=====
23758   *           get_filp
23759   *=====
23760 PUBLIC struct filp *get_filp(filid)
23761 int filid;          /* file descriptor */
23762 {
23763 /* See if 'filid' refers to a valid file descr.  If so, return its filp ptr. */
23765
23766     err_code = EBADF;
23767     if (filid < 0 || filid >= OPEN_MAX ) return(NIL_FILP);
23768     return(fp->fp_filp[filid]); /* may also be NIL_FILP */

```

```

23769 }
23770
23771 /*=====
23772 *          find_filp
23773 *=====
23774 PUBLIC struct filp *find_filp(register struct inode *rip, mode_t bits)
23775 {
23776 /* Find a filp slot that refers to the inode 'rip' in a way as described
23777 * by the mode bit 'bits'. Used for determining whether somebody is still
23778 * interested in either end of a pipe. Also used when opening a FIFO to
23779 * find partners to share a filp field with (to shared the file position).
23780 * Like 'get_fd' it performs its job by linear search through the filp table.
23781 */
23782
23783 register struct filp *f;
23784
23785 for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
23786     if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bits)){
23787         return(f);
23788     }
23789 }
23790
23791 /* If control passes here, the filp wasn't there. Report that back. */
23792 return(NIL_FILP);
23793 }
```

```
+++++
servers/fs/lock.c
+++++
```

```

23800 /* This file handles advisory file locking as required by POSIX.
23801 *
23802 * The entry points into this file are
23803 *   lock_op:   perform locking operations for FCNTL system call
23804 *   lock_revive: revive processes when a lock is released
23805 */
23806
23807 #include "fs.h"
23808 #include <minix/com.h>
23809 #include <fcntl.h>
23810 #include <unistd.h>
23811 #include "file.h"
23812 #include "fproc.h"
23813 #include "inode.h"
23814 #include "lock.h"
23815 #include "param.h"
23816
23817 /*=====
23818 *          lock_op
23819 *=====
23820 PUBLIC int lock_op(f, req)
23821 struct filp *f;
23822 int req;                      /* either F_SETLK or F_SETLKW */
23823 {
23824 /* Perform the advisory locking required by POSIX. */
23825
23826     int r, ltype, i, conflict = 0, unlocking = 0;
23827     mode_t mo;
23828     off_t first, last;
23829     struct flock flock;
23830     vir_bytes user_flock;
23831     struct file_lock *flp, *flp2, *empty;
23832
23833 /* Fetch the flock structure from user space. */
23834     user_flock = (vir_bytes) m_in.name1;
23835     r = sys_datacopy(who, (vir_bytes) user_flock,
23836                      FS_PROC_NR, (vir_bytes) &flock, (phys_bytes) sizeof(flock));
```

```

23837     if (r != OK) return(EINVAL);
23838
23839     /* Make some error checks. */
23840     ltype = flock.l_type;
23841     mo = f->filp_mode;
23842     if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
23843     if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
23844     if ((f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
23845     if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
23846     if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);
23847
23848     /* Compute the first and last bytes in the lock region. */
23849     switch (flock.l_whence) {
23850         case SEEK_SET: first = 0; break;
23851         case SEEK_CUR: first = f->filp_pos; break;
23852         case SEEK_END: first = f->filp_ino->i_size; break;
23853         default:      return(EINVAL);
23854     }
23855     /* Check for overflow. */
23856     if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
23857         return(EINVAL);
23858     if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
23859         return(EINVAL);
23860     first = first + flock.l_start;
23861     last = first + flock.l_len - 1;
23862     if (flock.l_len == 0) last = MAX_FILE_POS;
23863     if (last < first) return(EINVAL);
23864
23865     /* Check if this region conflicts with any existing lock. */
23866     empty = (struct file_lock *) 0;
23867     for (flp = &file_lock[0]; flp < & file_lock[NR_LOCKS]; flp++) {
23868         if (flp->lock_type == 0) {
23869             if (empty == (struct file_lock *) 0) empty = flp;
23870             continue; /* 0 means unused slot */
23871         }
23872         if (flp->lock_inode != f->filp_ino) continue; /* different file */
23873         if (last < flp->lock_first) continue; /* new one is in front */
23874         if (first > flp->lock_last) continue; /* new one is afterwards */
23875         if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
23876         if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;
23877
23878         /* There might be a conflict. Process it. */
23879         conflict = 1;
23880         if (req == F_GETLK) break;
23881
23882         /* If we are trying to set a lock, it just failed. */
23883         if (ltype == F_RDLCK || ltype == F_WRLCK) {
23884             if (req == F_SETLK) {
23885                 /* For F_SETLK, just report back failure. */
23886                 return(EAGAIN);
23887             } else {
23888                 /* For F_SETLKW, suspend the process. */
23889                 suspend(XLOCK);
23890                 return(SUSPEND);
23891             }
23892         }
23893
23894         /* We are clearing a lock and we found something that overlaps. */
23895         unlocking = 1;
23896         if (first <= flp->lock_first && last >= flp->lock_last) {
23897             flp->lock_type = 0; /* mark slot as unused */
23898             nr_locks--; /* number of locks is now 1 less */
23899             continue;
23900         }
23901
23902         /* Part of a locked region has been unlocked. */
23903         if (first <= flp->lock_first) {
23904             flp->lock_first = last + 1;

```

```

23905         continue;
23906     }
23907
23908     if (last >= ffp->lock_last) {
23909         ffp->lock_last = first - 1;
23910         continue;
23911     }
23912
23913     /* Bad luck. A lock has been split in two by unlocking the middle. */
23914     if (nr_locks == NR_LOCKS) return(ENOLCK);
23915     for (i = 0; i < NR_LOCKS; i++) {
23916         if (file_lock[i].lock_type == 0) break;
23917     ffp2 = &file_lock[i];
23918     ffp2->lock_type = ffp->lock_type;
23919     ffp2->lock_pid = ffp->lock_pid;
23920     ffp2->lock_inode = ffp->lock_inode;
23921     ffp2->lock_first = last + 1;
23922     ffp2->lock_last = ffp->lock_last;
23923     ffp->lock_last = first - 1;
23924     nr_locks++;
23925   }
23926   if (unlocking) lock_revive();
23927
23928   if (req == F_GETLK) {
23929     if (conflict) {
23930         /* GETLK and conflict. Report on the conflicting lock. */
23931         flock.l_type = ffp->lock_type;
23932         flock.l_whence = SEEK_SET;
23933         flock.l_start = ffp->lock_first;
23934         flock.l_len = ffp->lock_last - ffp->lock_first + 1;
23935         flock.l_pid = ffp->lock_pid;
23936
23937     } else {
23938         /* It is GETLK and there is no conflict. */
23939         flock.l_type = F_UNLCK;
23940     }
23941
23942     /* Copy the flock structure back to the caller. */
23943     r = sys_datacopy(FS_PROC_NR, (vir_bytes) &flock,
23944                     who, (vir_bytes) user_flock, (phys_bytes) sizeof(flock));
23945     return(r);
23946 }
23947
23948   if (ltype == F_UNLCK) return(OK);      /* unlocked a region with no locks */
23949
23950   /* There is no conflict. If space exists, store new lock in the table. */
23951   if (empty == (struct file_lock *) 0) return(ENOLCK); /* table full */
23952   empty->lock_type = ltype;
23953   empty->lock_pid = fp->fp_pid;
23954   empty->lock_inode = f->filp_ino;
23955   empty->lock_first = first;
23956   empty->lock_last = last;
23957   nr_locks++;
23958   return(OK);
23959 }

23960 /*=====
23961 *          lock_revive
23962 *=====
23963 PUBLIC void lock_revive()
23964 {
23965     /* Go find all the processes that are waiting for any kind of lock and
23966     * revive them all. The ones that are still blocked will block again when
23967     * they run. The others will complete. This strategy is a space-time
23968     * tradeoff. Figuring out exactly which ones to unblock now would take
23969     * extra code, and the only thing it would win would be some performance in
23970     * extremely rare circumstances (namely, that somebody actually used

```

```

23972 * locking).
23973 */
23974
23975 int task;
23976 struct fproc *fptr;
23977
23978 for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
23979     task = -fptr->fp_task;
23980     if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
23981         revive( (int) (fptr - fproc), 0);
23982     }
23983 }
23984 }

+++++
servers/fs/main.c
++++

24000 /* This file contains the main program of the File System. It consists of
24001 * a loop that gets messages requesting work, carries out the work, and sends
24002 * replies.
24003 *
24004 * The entry points into this file are:
24005 *   main:      main program of the File System
24006 *   reply:    send a reply to a process after the requested work is done
24007 *
24008 */
24009
24010 struct super_block;           /* proto.h needs to know this */
24011
24012 #include "fs.h"
24013 #include <fcntl.h>
24014 #include <string.h>
24015 #include <stdio.h>
24016 #include <signal.h>
24017 #include <stdlib.h>
24018 #include <sys/ioc_memory.h>
24019 #include <sys/svrctl.h>
24020 #include <minix/callnr.h>
24021 #include <minix/com.h>
24022 #include <minix/keymap.h>
24023 #include <minix/const.h>
24024 #include "buf.h"
24025 #include "file.h"
24026 #include "fproc.h"
24027 #include "inode.h"
24028 #include "param.h"
24029 #include "super.h"
24030
24031 FORWARD _PROTOTYPE( void fs_init, (void) );
24032 FORWARD _PROTOTYPE( int igetenv, (char *var, int optional) );
24033 FORWARD _PROTOTYPE( void get_work, (void) );
24034 FORWARD _PROTOTYPE( void load_ram, (void) );
24035 FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev) );
24036
24037 /*=====
24038 *                         main
24039 *=====
24040 PUBLIC int main()
24041 {
24042     /* This is the main program of the file system. The main loop consists of
24043     * three major activities: getting new work, processing the work, and sending
24044     * the reply. This loop never terminates as long as the file system runs.
24045     */
24046     sigset_t sigset;
24047     int error;

```

```

24048
24049     fs_init();
24050
24051     /* This is the main loop that gets work, processes it, and sends replies. */
24052     while (TRUE) {
24053         get_work();           /* sets who and call_nr */

24054         fp = &fproc[who];      /* pointer to proc table struct */
24055         super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE); /* su? */

24056
24057         /* Check for special control messages first. */
24058         if (call_nr == SYS_SIG) {
24059             sigset = m_in.NOTIFY_ARG;
24060             if (sigismember(&sigset, SIGKSTOP)) {
24061                 do_sync();
24062                 sys_exit(0);        /* never returns */
24063             }
24064         } else if (call_nr == SYN_ALARM) {
24065             /* Not a user request; system has expired one of our timers,
24066              * currently only in use for select(). Check it.
24067              */
24068             fs_expire_timers(m_in.NOTIFY_TIMESTAMP);
24069         } else if ((call_nr & NOTIFY_MESSAGE)) {
24070             /* Device notifies us of an event. */
24071             dev_status(&m_in);
24072         } else {
24073             /* Call the internal function that does the work. */
24074             if (call_nr < 0 || call_nr >= NCALLS) {
24075                 error = ENOSYS;
24076                 printf("FS,warning illegal %d system call by %d\n",call_nr,who);
24077             } else if (fp->fp_pid == PID_FREE) {
24078                 error = ENOSYS;
24079                 printf("FS, bad process, who = %d, call_nr = %d, slot1 = %d\n",
24080                         who, call_nr, m_in.slot1);
24081             } else {
24082                 error = (*call_vec[call_nr])();
24083             }
24084
24085             /* Copy the results back to the user and send reply. */
24086             if (error != SUSPEND) { reply(who, error); }
24087             if (rdahed_inode != NIL_INODE) {
24088                 read_ahead(); /* do block read ahead */
24089             }
24090         }
24091     }
24092     return(OK);           /* shouldn't come here */
24093 }
24094 }

24095 /*=====
24096 *          get_work
24097 *=====
24098 */
24099 PRIVATE void get_work()
24100 {
24101     /* Normally wait for new input. However, if 'reviving' is
24102     * nonzero, a suspended process must be awakened.
24103     */
24104     register struct fproc *rp;
24105
24106     if (reviving != 0) {
24107         /* Revive a suspended process. */
24108         for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
24109             if (rp->fp_revived == REVIVING) {
24110                 who = (int)(rp - fproc);
24111                 call_nr = rp->fp_fd & BYTE;
24112                 m_in.fd = (rp->fp_fd >>8) & BYTE;
24113                 m_in.buffer = rp->fp_buffer;
24114                 m_in.nbytes = rp->fp_nbytes;

```

```

24115                     rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
24116                     rp->fp_revived = NOT_REVIVING;
24117                     reviving--;
24118                     return;
24119                 }
24120             panic(__FILE__,"get_work couldn't revive anyone", NO_NUM);
24121         }
24122
24123     /* Normal case. No one to revive. */
24124     if (receive(ANY, &m_in) != OK) panic(__FILE__,"fs receive error", NO_NUM);
24125     who = m_in.m_source;
24126     call_nr = m_in.m_type;
24127 }

24129 /*=====
24130 *                                buf_pool
24131 *=====
24132 PRIVATE void buf_pool(void)
24133 {
24134     /* Initialize the buffer pool. */
24135
24136     register struct buf *bp;
24137
24138     bufs_in_use = 0;
24139     front = &buf[0];
24140     rear = &buf[NR_BUFS - 1];
24141
24142     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
24143         bp->b_blocknr = NO_BLOCK;
24144         bp->b_dev = NO_DEV;
24145         bp->b_next = bp + 1;
24146         bp->b_prev = bp - 1;
24147     }
24148     buf[0].b_prev = NIL_BUF;
24149     buf[NR_BUFS - 1].b_next = NIL_BUF;
24150
24151     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
24152     buf_hash[0] = front;
24153 }
24154 }

24156 /*=====
24157 *                                reply
24158 *=====
24159 PUBLIC void reply(whom, result)
24160 int whom;                      /* process to reply to */
24161 int result;                     /* result of the call (usually OK or error #) */
24162 {
24163     /* Send a reply to a user process. It may fail (if the process has just
24164     * been killed by a signal), so don't check the return code. If the send
24165     * fails, just ignore it.
24166     */
24167     int s;
24168     m_out.reply_type = result;
24169     s = send(whom, &m_out);
24170     if (s != OK) printf("FS: couldn't send reply %d: %d\n", result, s);
24171 }

24173 /*=====
24174 *                                fs_init
24175 *=====
24176 PRIVATE void fs_init()
24177 {
24178     /* Initialize global variables, tables, etc. */
24179     register struct inode *rip;
24180     register struct fproc *rfp;
24181     message mess;

```

```

24182     int s;
24183
24184     /* Initialize the process table with help of the process manager messages.
24185      * Expect one message for each system process with its slot number and pid.
24186      * When no more processes follow, the magic process number NONE is sent.
24187      * Then, stop and synchronize with the PM.
24188      */
24189     do {
24190         if (OK != (s=receive(PM_PROC_NR, &mess)))
24191             panic(__FILE__,"FS couldn't receive from PM", s);
24192         if (NONE == mess.PR_PROC_NR) break;
24193
24194         rfp = &fproc[mess.PR_PROC_NR];
24195         rfp->fp_pid = mess.PR_PID;
24196         rfp->fp_realuid = (uid_t) SYS_UID;
24197         rfp->fp_effuid = (uid_t) SYS_UID;
24198         rfp->fp_realgid = (gid_t) SYS_GID;
24199         rfp->fp_effgid = (gid_t) SYS_GID;
24200         rfp->fp_umask = ~0;
24201
24202     } while (TRUE);           /* continue until process NONE */
24203     mess.m_type = OK;          /* tell PM that we succeeded */
24204     s=send(PM_PROC_NR, &mess); /* send synchronization message */
24205
24206     /* All process table entries have been set. Continue with FS initialization.
24207      * Certain relations must hold for the file system to work at all. Some
24208      * extra block_size requirements are checked at super-block-read-in time.
24209      */
24210     if (OPEN_MAX > 127) panic(__FILE__,"OPEN_MAX > 127", NO_NUM);
24211     if (NR_BUFS < 6) panic(__FILE__,"NR_BUFS < 6", NO_NUM);
24212     if (V1_INODE_SIZE != 32) panic(__FILE__,"V1 inode size != 32", NO_NUM);
24213     if (V2_INODE_SIZE != 64) panic(__FILE__,"V2 inode size != 64", NO_NUM);
24214     if (OPEN_MAX > 8 * sizeof(long))
24215         panic(__FILE__,"Too few bits in fp_cloexec", NO_NUM);
24216
24217     /* The following initializations are needed to let dev_opcl succeed .*/
24218     fp = (struct fproc *) NULL;
24219     who = FS_PROC_NR;
24220
24221     buf_pool();           /* initialize buffer pool */
24222     build_dmap();          /* build device table and map boot driver */
24223     load_ram();            /* init RAM disk, load if it is root */
24224     load_super(root_dev); /* load super block for root device */
24225     init_select();          /* init select() structures */
24226
24227     /* The root device can now be accessed; set process directories. */
24228     for (rfp=&fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
24229         if (rfp->fp_pid != PID_FREE) {
24230             rip = get_inode(root_dev, ROOT_INODE);
24231             dup_inode(rip);
24232             rfp->fp_rootdir = rip;
24233             rfp->fp_workdir = rip;
24234         }
24235     }
24236 }

24238 /*=====
24239 *          igetenv
24240 *=====
24241 PRIVATE int igetenv(key, optional)
24242 char *key;
24243 int optional;
24244 {
24245 /* Ask kernel for an integer valued boot environment variable. */
24246     char value[64];
24247     int i;
24248

```

```

24249     if ((i = env_get_param(key, value, sizeof(value))) != OK) {
24250         if (!optional)
24251             printf("FS: Warning, couldn't get monitor param: %d\n", i);
24252         return 0;
24253     }
24254     return atoi(value);
24255 }

24257 /*=====
24258 *          load_ram
24259 *=====
24260 PRIVATE void load_ram(void)
24261 {
24262 /* Allocate a RAM disk with size given in the boot parameters. If a RAM disk
24263 * image is given, the copy the entire image device block-by-block to a RAM
24264 * disk with the same size as the image.
24265 * If the root device is not set, the RAM disk will be used as root instead.
24266 */
24267 register struct buf *bp, *bp1;
24268 u32_t lcount, ram_size_kb;
24269 zone_t zones;
24270 struct super_block *sp, *dsp;
24271 block_t b;
24272 Dev_t image_dev;
24273 static char sbbuf[MIN_BLOCK_SIZE];
24274 int block_size_image, block_size_ram, ramfs_block_size;
24275 int s;
24276
24277 /* Get some boot environment variables. */
24278 root_dev = getenv("rootdev", 0);
24279 image_dev = getenv("ramimagedev", 0);
24280 ram_size_kb = getenv("ramsize", 0);
24281
24282 /* Open the root device. */
24283 if (dev_open(root_dev, FS_PROC_NR, R_BIT|W_BIT) != OK)
24284     panic(FILE__, "Cannot open root device", NO_NUM);
24285
24286 /* If we must initialize a ram disk, get details from the image device. */
24287 if (root_dev == DEV_RAM) {
24288     u32_t fsmax, probudev;
24289
24290     /* If we are running from CD, see if we can find it. */
24291     if (getenv("cdproberoot", 1) && (probudev=cdprobe()) != NO_DEV) {
24292         char devnum[10];
24293         struct sysgetenv env;
24294
24295         /* If so, this is our new RAM image device. */
24296         image_dev = probudev;
24297
24298         /* Tell PM about it, so userland can find out about it
24299          * with sysenv interface.
24300          */
24301         env.key = "cdproberoot";
24302         env.keylen = strlen(env.key);
24303         sprintf(devnum, "%d", (int) probudev);
24304         env.val = devnum;
24305         env.vallen = strlen(devnum);
24306         svrctl(MMSETPARAM, &env);
24307     }
24308
24309     /* Open image device for RAM root. */
24310     if (dev_open(image_dev, FS_PROC_NR, R_BIT) != OK)
24311         panic(FILE__, "Cannot open RAM image device", NO_NUM);
24312
24313     /* Get size of RAM disk image from the super block. */
24314     sp = &super_block[0];
24315     sp->s_dev = image_dev;

```

```

24316     if (read_super(sp) != OK)
24317         panic(__FILE__,"Bad RAM disk image FS", NO_NUM);
24318
24319     lcount = sp->s_zones << sp->s_log_zone_size; /* # blks on root dev*/
24320
24321     /* Stretch the RAM disk file system to the boot parameters size, but
24322      * no further than the last zone bit map block allows.
24323      */
24324     if (ram_size_kb*1024 < lcount*sp->s_block_size)
24325         ram_size_kb = lcount*sp->s_block_size/1024;
24326     fsmax = (u32_t) sp->s_zmap_blocks * CHAR_BIT * sp->s_block_size;
24327     fsmax = (fsmax + (sp->s_firstdatazone-1)) << sp->s_log_zone_size;
24328     if (ram_size_kb*1024 > fsmax*sp->s_block_size)
24329         ram_size_kb = fsmax*sp->s_block_size/1024;
24330 }
24331
24332 /* Tell RAM driver how big the RAM disk must be. */
24333 m_out.m_type = DEV_IOCTL;
24334 m_out.PROC_NR = FS_PROC_NR;
24335 m_out.DEVICE = RAM_DEV;
24336 m_out.REQUEST = MIOCRAMSIZE; /* I/O control to use */
24337 m_out.POSITION = (ram_size_kb * 1024); /* request in bytes */
24338 if ((s=sendrec(MEM_PROC_NR, &m_out)) != OK)
24339     panic("FS","sendrec from MEM failed", s);
24340 else if (m_out.REP_STATUS != OK) {
24341     /* Report and continue, unless RAM disk is required as root FS. */
24342     if (root_dev != DEV_RAM) {
24343         report("FS","can't set RAM disk size", m_out.REP_STATUS);
24344         return;
24345     } else {
24346         panic(__FILE__,"can't set RAM disk size", m_out.REP_STATUS);
24347     }
24348 }
24349
24350 /* See if we must load the RAM disk image, otherwise return. */
24351 if (root_dev != DEV_RAM)
24352     return;
24353
24354 /* Copy the blocks one at a time from the image to the RAM disk. */
24355 printf("Loading RAM disk onto /dev/ram:\33[23CLoaded: 0 KB");
24356
24357 inode[0].i_mode = I_BLOCK_SPECIAL; /* temp inode for rahead() */
24358 inode[0].i_size = LONG_MAX;
24359 inode[0].i_dev = image_dev;
24360 inode[0].i_zone[0] = image_dev;
24361
24362 block_size_ram = get_block_size(DEV_RAM);
24363 block_size_image = get_block_size(image_dev);
24364
24365 /* RAM block size has to be a multiple of the root image block
24366   * size to make copying easier.
24367   */
24368 if (block_size_image % block_size_ram) {
24369     printf("\nram block size: %d image block size: %d\n",
24370           block_size_ram, block_size_image);
24371     panic(__FILE__,"ram disk block size must be a multiple of "
24372           "the image disk block size", NO_NUM);
24373 }
24374
24375 /* Loading blocks from image device. */
24376 for (b = 0; b < (block_t) lcount; b++) {
24377     int rb, factor;
24378     bp = rahead(&inode[0], b, (off_t)block_size_image * b, block_size_image);
24379     factor = block_size_image/block_size_ram;
24380     for(rb = 0; rb < factor; rb++) {
24381         bp1 = get_block(root_dev, b * factor + rb, NO_READ);
24382         memcpy(bp1->b_data, bp->b_data + rb * block_size_ram,

```



```

24450     dup_inode(rip);
24451     sp->s_isup = rip;
24452     sp->s_rd_only = 0;
24453     return;
24454 }

+++++servers/fs/open.c+++++
+++++servers/fs/open.c+++++

24500 /* This file contains the procedures for creating, opening, closing, and
24501 * seeking on files.
24502 *
24503 * The entry points into this file are
24504 *   do_creat: perform the CREAT system call
24505 *   do_open:  perform the OPEN system call
24506 *   do_mknod: perform the MKNOD system call
24507 *   do_mkdir: perform the MKDIR system call
24508 *   do_close: perform the CLOSE system call
24509 *   do_lseek: perform the LSEEK system call
24510 */
24511
24512 #include "fs.h"
24513 #include <sys/stat.h>
24514 #include <fcntl.h>
24515 #include <minix/callnr.h>
24516 #include <minix/com.h>
24517 #include "buf.h"
24518 #include "file.h"
24519 #include "fproc.h"
24520 #include "inode.h"
24521 #include "lock.h"
24522 #include "param.h"
24523 #include "super.h"
24524
24525 #define offset m2_ll1
24526
24527 PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};
24528
24529 FORWARD _PROTOTYPE( int common_open, (int oflags, mode_t omode) );
24530 FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip, mode_t bits, int oflags));
24531 FORWARD _PROTOTYPE( struct inode *new_node, (char *path, mode_t bits,
24532                                     zone_t z0) );
24533
24534 /*=====
24535 *          do_creat
24536 *=====
24537 PUBLIC int do_creat()
24538 {
24539 /* Perform the creat(name, mode) system call. */
24540     int r;
24541
24542     if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
24543     r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) m_in.mode);
24544     return(r);
24545 }

24546 /*=====
24547 *          do_open
24548 *=====
24549 PUBLIC int do_open()
24550 {
24551 /* Perform the open(name, flags,...) system call. */
24552     int create_mode = 0;           /* is really mode_t but this gives problems */
24553     int r;
24554

```

```

24557 /* If O_CREAT is set, open has three parameters, otherwise two. */
24558 if (m_in.mode & O_CREAT) {
24559     create_mode = m_in.c_mode;
2460     r = fetch_name(m_in.c_name, m_in.name1_length, M1);
2461 } else {
2462     r = fetch_name(m_in.name, m_in.name_length, M3);
2463 }
2464
2465 if (r != OK) return(err_code); /* name was bad */
2466 r = common_open(m_in.mode, create_mode);
2467 return(r);
2468 }

24570 /*=====
24571 *          common_open
24572 *=====
24573 PRIVATE int common_open(register int oflags, mode_t omode)
24574 {
24575 /* Common code from do_creat and do_open. */
24576
24577     register struct inode *rip;
24578     int r, b, exist = TRUE;
24579     dev_t dev;
24580     mode_t bits;
24581     off_t pos;
24582     struct filp *filp_ptr, *filp2;
24583
24584     /* Remap the bottom two bits of oflags. */
24585     bits = (mode_t) mode_map[oflags & O_ACCMODE];
24586
24587     /* See if file descriptor and filp slots are available. */
24588     if ((r = get_fd(0, bits, &m_in.fd, &filp_ptr)) != OK) return(r);
24589
24590     /* If O_CREATE is set, try to make the file. */
24591     if (oflags & O_CREAT) {
24592         /* Create a new inode by calling new_node(). */
24593         omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
24594         rip = new_node(user_path, omode, NO_ZONE);
24595         r = err_code;
24596         if (r == OK) exist = FALSE; /* we just created the file */
24597         else if (r != EXIST) return(r); /* other error */
24598         else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
24599                                         flag is set this is an error */
24600     } else {
24601         /* Scan path name. */
24602         if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
24603     }
24604
24605     /* Claim the file descriptor and filp slot and fill them in. */
24606     fp->fp_filp[m_in.fd] = filp_ptr;
24607     filp_ptr->filp_count = 1;
24608     filp_ptr->filp_ino = rip;
24609     filp_ptr->filp_flags = oflags;
24610
24611     /* Only do the normal open code if we didn't just create the file. */
24612     if (exist) {
24613         /* Check protections. */
24614         if ((r = forbidden(rip, bits)) == OK) {
24615             /* Opening reg. files directories and special files differ. */
24616             switch (rip->i_mode & I_TYPE) {
24617                 case I_REGULAR:
24618                     /* Truncate regular file if O_TRUNC. */
24619                     if (oflags & O_TRUNC) {
24620                         if ((r = forbidden(rip, W_BIT)) !=OK) break;
24621                         truncate(rip);
24622                         wipe_inode(rip);
24623                         /* Send the inode from the inode cache to the

```

```

24624 * block cache, so it gets written on the next
24625 * cache flush.
24626 */
24627 rw_inode(rip, WRITING);
24628 }
24629 break;
24630
24631 case I_DIRECTORY:
24632 /* Directories may be read but not written. */
24633 r = (bits & W_BIT ? EISDIR : OK);
24634 break;
24635
24636 case I_CHAR_SPECIAL:
24637 case I_BLOCK_SPECIAL:
24638 /* Invoke the driver for special processing. */
24639 dev = (dev_t) rip->i_zone[0];
24640 r = dev_open(dev, who, bits | (oflags & ~O_ACCMODE));
24641 break;
24642
24643 case I_NAMED_PIPE:
24644 oflags |= O_APPEND; /* force append mode */
24645 fil_ptr->filp_flags = oflags;
24646 r = pipe_open(rip, bits, oflags);
24647 if (r != ENXIO) {
24648 /* See if someone else is doing a rd or wt on
24649 * the FIFO. If so, use its filp entry so the
24650 * file position will be automatically shared.
24651 */
24652 b = (bits & R_BIT ? R_BIT : W_BIT);
24653 fil_ptr->filp_count = 0; /* don't find self */
24654 if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
24655 /* Co-reader or writer found. Use it.*/
24656 fp->fp_filp[m_in.fd] = filp2;
24657 filp2->filp_count++;
24658 filp2->filp_ino = rip;
24659 filp2->filp_flags = oflags;
24660
24661 /* i_count was incremented incorrectly
24662 * by eatpath above, not knowing that
24663 * we were going to use an existing
24664 * filp entry. Correct this error.
24665 */
24666 rip->i_count--;
24667 } else {
24668 /* Nobody else found. Restore filp. */
24669 fil_ptr->filp_count = 1;
24670 if (b == R_BIT)
24671 pos = rip->i_zone[V2_NR_DZONES+0];
24672 else
24673 pos = rip->i_zone[V2_NR_DZONES+1];
24674 fil_ptr->filp_pos = pos;
24675 }
24676 }
24677 break;
24678 }
24679 }
24680
24681 /* If error, release inode. */
24682 if (r != OK) {
24683 if (r == SUSPEND) return(r); /* Oops, just suspended */
24684 fp->fp_filp[m_in.fd] = NIL_FILP;
24685 fil_ptr->filp_count= 0;
24686 put_inode(rip);
24687 return(r);
24688 }
24689
24690

```

```

24691     return(_in.fd);
24692 }

24694 /*=====
24695 *          new_node
24696 *=====
24697 PRIVATE struct inode *new_node(char *path, mode_t bits, zone_t z0)
24698 {
24699 /* New_node() is called by common_open(), do_mknod(), and do_mkdir().
24700 * In all cases it allocates a new inode, makes a directory entry for it on
24701 * the path 'path', and initializes it. It returns a pointer to the inode if
24702 * it can do this; otherwise it returns NIL_INODE. It always sets 'err_code'
24703 * to an appropriate value (OK or an error code).
24704 */
24705
24706     register struct inode *rlast_dir_ptr, *rip;
24707     register int r;
24708     char string[NAME_MAX];
24709
24710 /* See if the path can be opened down to the last directory. */
24711 if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);
24712
24713 /* The final directory is accessible. Get final component of the path. */
24714 rip = advance(rlast_dir_ptr, string);
24715 if (rip == NIL_INODE && err_code == ENOENT) {
24716     /* Last path component does not exist. Make new directory entry. */
24717     if ((rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
24718         /* Can't creat new inode: out of inodes. */
24719         put_inode(rlast_dir_ptr);
24720         return(NIL_INODE);
24721     }
24722
24723     /* Force inode to the disk before making directory entry to make
24724     * the system more robust in the face of a crash: an inode with
24725     * no directory entry is much better than the opposite.
24726     */
24727     rip->i_nlinks++;
24728     rip->i_zone[0] = z0;           /* major/minor device numbers */
24729     rw_inode(rip, WRITING);      /* force inode to disk now */
24730
24731 /* New inode acquired. Try to make directory entry. */
24732 if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num, ENTER)) != OK) {
24733     put_inode(rlast_dir_ptr);
24734     rip->i_nlinks--;           /* pity, have to free disk inode */
24735     rip->i_dirt = DIRTY;       /* dirty inodes are written out */
24736     put_inode(rip); /* this call frees the inode */
24737     err_code = r;
24738     return(NIL_INODE);
24739 }
24740
24741 } else {
24742     /* Either last component exists, or there is some problem. */
24743     if (rip != NIL_INODE)
24744         r = EEXIST;
24745     else
24746         r = err_code;
24747 }
24748
24749 /* Return the directory inode and exit. */
24750 put_inode(rlast_dir_ptr);
24751 err_code = r;
24752 return(rip);
24753 }

24755 /*=====
24756 *          pipe_open
24757 *=====
24758 PRIVATE int pipe_open(register struct inode *rip, register mode_t bits,

```

```

24759         register int oflags)
24600     {
2461 /* This function is called from common_open. It checks if
2462 * there is at least one reader/writer pair for the pipe, if not
2463 * it suspends the caller, otherwise it revives all other blocked
2464 * processes hanging on the pipe.
2465 */
2466
2467     rip->i_pipe = I_PIPE;
2468     if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
2469         if (oflags & O_NONBLOCK) {
2470             if (bits & W_BIT) return(ENXIO);
2471         } else {
2472             suspend(XPOPEN); /* suspend caller */
2473             return(SUSPEND);
2474         }
2475     } else if (susp_count > 0) {/* revive blocked processes */
2476         release(rip, OPEN, susp_count);
2477         release(rip, CREAT, susp_count);
2478     }
2479     return(OK);
2480 }

2481 /*=====
2482 *          do_mknod
2483 *=====
2484 */
2485 PUBLIC int do_mknod()
2486 {
2487 /* Perform the mknod(name, mode, addr) system call. */

2488     register mode_t bits, mode_bits;
2489     struct inode *ip;
2490
2491     /* Only the super_user may make nodes other than fifos. */
2492     mode_bits = (mode_t) m_in.mk_mode; /* mode of the inode */
2493     if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
2494     if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
2495     bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
2496     ip = new_node(user_path, bits, (zone_t) m_in.mk_z0);
2497     put_inode(ip);
2498     return(err_code);
2499 }
2500

2501 /*=====
2502 *          do_mkdir
2503 *=====
2504 */
2505 PUBLIC int do_mkdir()
2506 {
2507 /* Perform the mkdir(name, mode) system call. */

2508     int r1, r2;           /* status codes */
2509     ino_t dot, dotdot;    /* inode numbers for . and .. */
2510     mode_t bits;          /* mode bits for the new inode */
2511     char string[NAME_MAX]; /* last component of the new dir's path name */
2512     register struct inode *rip, *ldirp;
2513
2514     /* Check to see if it is possible to make another link in the parent dir. */
2515     if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
2516     ldirp = last_dir(user_path, string); /* pointer to new dir's parent */
2517     if (ldirp == NIL_INODE) return(err_code);
2518     if (ldirp->i_nlinks >= (ldirp->i_sp->s_version == V1 ?
2519         CHAR_MAX : SHRT_MAX)) {
2520         put_inode(ldirp); /* return parent */
2521         return(EMLINK);
2522     }
2523
2524     /* Next make the inode. If that fails, return error code. */
2525 }
```

```

24826 bits = I_DIRECTORY | (m_in.mode & RWX_MODES & fp->fp_umask);
24827 rip = new_node(user_path, bits, (zone_t) 0);
24828 if (rip == NIL_INODE || err_code == EEXIST) {
24829     put_inode(rip);           /* can't make dir: it already exists */
24830     put_inode(ldirp);        /* return parent too */
24831     return(err_code);
24832 }
24833
24834 /* Get the inode numbers for . and .. to enter in the directory. */
24835 dotdot = ldirp->i_num;      /* parent's inode number */
24836 dot = rip->i_num;          /* inode number of the new dir itself */
24837
24838 /* Now make dir entries for . and .. unless the disk is completely full. */
24839 /* Use dot1 and dot2, so the mode of the directory isn't important. */
24840 rip->i_mode = bits;         /* set mode */
24841 r1 = search_dir(rip, dot1, &dot, ENTER);      /* enter . in the new dir */
24842 r2 = search_dir(rip, dot2, &dotdot, ENTER);    /* enter .. in the new dir */
24843
24844 /* If both . and .. were successfully entered, increment the link counts. */
24845 if (r1 == OK && r2 == OK) {
24846     /* Normal case. It was possible to enter . and .. in the new dir. */
24847     rip->i_nlinks++;        /* this accounts for . */
24848     ldirp->i_nlinks++;     /* this accounts for .. */
24849     ldirp->i_dirt = DIRTY; /* mark parent's inode as dirty */
24850 } else {
24851     /* It was not possible to enter . or .. probably disk was full. */
24852     (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
24853     rip->i_nlinks--;       /* undo the increment done in new_node() */
24854 }
24855 rip->i_dirt = DIRTY;        /* either way, i_nlinks has changed */
24856
24857 put_inode(ldirp);          /* return the inode of the parent dir */
24858 put_inode(rip);            /* return the inode of the newly made dir */
24859 return(err_code);          /* new_node() always sets 'err_code' */
24860 }

24861 /*=====
24862 *          do_close
24863 *=====
24864 PUBLIC int do_close()
24865 {
24866 /* Perform the close(fd) system call. */
24867
24868 register struct filp *rfilp;
24869 register struct inode *rip;
24870 struct file_lock *flp;
24871 int rw, mode_word, lock_count;
24872 dev_t dev;
24873
24874 /* First locate the inode that belongs to the file descriptor. */
24875 if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
24876 rip = rfilp->filp_ino;      /* 'rip' points to the inode */
24877
24878 if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
24879     /* Check to see if the file is special. */
24880     mode_word = rip->i_mode & I_TYPE;
24881     if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
24882         dev = (dev_t) rip->i_zone[0];
24883         if (mode_word == I_BLOCK_SPECIAL) {
24884             /* Invalidate cache entries unless special is mounted
24885              * or ROOT
24886              */
24887             if (!mounted(rip)) {
24888                 (void) do_sync();           /* purge cache */
24889                 invalidate(dev);
24890             }
24891     }

```

```

24893             /* Do any special processing on device close. */
24894             dev_close(dev);
24895         }
24896     }
24897
24898     /* If the inode being closed is a pipe, release everyone hanging on it. */
24899     if (rip->i_pipe == I_PIPE) {
24900         rw = (rfile->filp_mode & R_BIT ? WRITE : READ);
24901         release(rip, rw, NR_PROCS);
24902     }
24903
24904     /* If a write has been done, the inode is already marked as DIRTY. */
24905     if (--rfile->filp_count == 0) {
24906         if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
24907             /* Save the file position in the i-node in case needed later.
24908             * The read and write positions are saved separately. The
24909             * last 3 zones in the i-node are not used for (named) pipes.
24910             */
24911             if (rfile->filp_mode == R_BIT)
24912                 rip->i_zone[V2_NR_DZONES+0] = (zone_t) rfile->filp_pos;
24913             else
24914                 rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfile->filp_pos;
24915         }
24916         put_inode(rip);
24917     }
24918
24919     fp->fp_cloexec &= ~(1L << m_in.fd); /* turn off close-on-exec bit */
24920     fp->fp_filp[m_in.fd] = NIL_FILP;
24921
24922     /* Check to see if the file is locked. If so, release all locks. */
24923     if (nr_locks == 0) return(OK);
24924     lock_count = nr_locks; /* save count of locks */
24925     for (fip = &file_lock[0]; fip < &file_lock[NR_LOCKS]; fip++) {
24926         if (fip->lock_type == 0) continue; /* slot not in use */
24927         if (fip->lock_inode == rip && fip->lock_pid == fp->fp_pid) {
24928             fip->lock_type = 0;
24929             nr_locks--;
24930         }
24931     }
24932     if (nr_locks < lock_count) lock_revive(); /* lock released */
24933     return(OK);
24934 }

24935 /*=====
24936      *
24937      *          do_lseek
24938      *=====
24939 PUBLIC int do_lseek()
24940 {
24941     /* Perform the lseek(ls_fd, offset, whence) system call. */
24942
24943     register struct filp *rfile;
24944     register off_t pos;
24945
24946     /* Check to see if the file descriptor is valid. */
24947     if ( (rfile = get_filp(m_in.ls_fd)) == NIL_FILP) return(err_code);
24948
24949     /* No lseek on pipes. */
24950     if (rfile->filp_ino->i_pipe == I_PIPE) return(ESPIPE);
24951
24952     /* The value of 'whence' determines the start position to use. */
24953     switch(m_in.whence) {
24954         case 0: pos = 0; break;
24955         case 1: pos = rfile->filp_pos; break;
24956         case 2: pos = rfile->filp_ino->i_size; break;
24957         default: return(EINVAL);
24958     }
24959

```

```

24960 /* Check for overflow. */
24961 if (((long)m_in.offset > 0) && ((long)(pos + m_in.offset) < (long)pos))
24962     return(EINVAL);
24963 if (((long)m_in.offset < 0) && ((long)(pos + m_in.offset) > (long)pos))
24964     return(EINVAL);
24965 pos = pos + m_in.offset;
24966
24967 if (pos != rfilp->filp_pos)
24968     rfilp->filp_ino->i_seek = ISEEK;           /* inhibit read ahead */
24969 rfilp->filp_pos = pos;
24970 m_out.reply_ll1 = pos;           /* insert the long into the output message */
24971 return(OK);
24972 }

```

```

+++++
servers/fs/read.c
+++++

```

```

25000 /* This file contains the heart of the mechanism used to read (and write)
25001 * files. Read and write requests are split up into chunks that do not cross
25002 * block boundaries. Each chunk is then processed in turn. Reads on special
25003 * files are also detected and handled.
25004 *
25005 * The entry points into this file are
25006 *   do_read:    perform the READ system call by calling read_write
25007 *   read_write: actually do the work of READ and WRITE
25008 *   read_map:   given an inode and file position, look up its zone number
25009 *   rd_indir:   read an entry in an indirect block
25010 *   read_ahead: manage the block read ahead business
25011 */
25012
25013 #include "fs.h"
25014 #include <fcntl.h>
25015 #include <minix/com.h>
25016 #include "buf.h"
25017 #include "file.h"
25018 #include "fproc.h"
25019 #include "inode.h"
25020 #include "param.h"
25021 #include "super.h"
25022
25023 FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
25024             unsigned off, int chunk, unsigned left, int rw_flag,
25025             char *buff, int seg, int usr, int block_size, int *completed));
25026
25027 /*=====
25028 *                      do_read
25029 *=====
25030 PUBLIC int do_read()
25031 {
25032     return(read_write(READING));
25033 }
25034
25035 /*=====
25036 *                      read_write
25037 *=====
25038 PUBLIC int read_write(rw_flag)
25039 int rw_flag;           /* READING or WRITING */
25040 {
25041 /* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */
25042
25043     register struct inode *rip;
25044     register struct filp *f;

```

```

25045     off_t bytes_left, f_size, position;
25046     unsigned int off, cum_io;
25047     int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
25048     int regular, partial_pipe = 0, partial_cnt = 0;
25049     mode_t mode_word;
25050     struct filp *wf;
25051     int block_size;
25052     int completed, r2 = OK;
25053     phys_bytes p;
25054
25055     /* left unfinished rw_chunk()s from previous call! this can't happen.
25056      * it means something has gone wrong we can't repair now.
25057      */
25058     if (bufs_in_use < 0) {
25059         panic(__FILE__,"start - bufs_in_use negative", bufs_in_use);
25060     }
25061
25062     /* MM loads segments by putting funny things in upper 10 bits of 'fd'. */
25063     if (who == PM_PROC_NR && (m_in.fd & (~BYTE)) ) {
25064         usr = m_in.fd >> 7;
25065         seg = (m_in.fd >> 5) & 03;
25066         m_in.fd &= 037;           /* get rid of user and segment bits */
25067     } else {
25068         usr = who;             /* normal case */
25069         seg = D;
25070     }
25071
25072     /* If the file descriptor is valid, get the inode, size and mode. */
25073     if (m_in.nbytes < 0) return(EINVAL);
25074     if ((f = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
25075     if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
25076         return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
25077     }
25078     if (m_in.nbytes == 0)
25079         return(0);          /* so char special files need not check for 0*/
25080
25081     /* check if user process has the memory it needs.
25082      * if not, copying will fail later.
25083      * do this after 0-check above because umap doesn't want to map 0 bytes.
25084      */
25085     if ((r = sys_umap(usr, seg, (vir_bytes) m_in.buffer, m_in.nbytes, &p)) != OK)
25086         return r;
25087     position = f->filp_pos;
25088     oflags = f->filp_flags;
25089     rip = f->filp_ino;
25090     f_size = rip->i_size;
25091     r = OK;
25092     if (rip->i_pipe == I_PIPE) {
25093         /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
25094         cum_io = fp->fp_cum_io_partial;
25095     } else {
25096         cum_io = 0;
25097     }
25098     op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
25099     mode_word = rip->i_mode & I_TYPE;
25100     regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;
25101
25102     if ((char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0))) {
25103         if (rip->i_zone[0] == NO_DEV)
25104             panic(__FILE__,"read_write tries to read from "
25105                   "character device NO_DEV", NO_NUM);
25106         block_size = get_block_size(rip->i_zone[0]);
25107     }
25108     if ((block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0))) {
25109         f_size = ULONG_MAX;
25110         if (rip->i_zone[0] == NO_DEV)
25111             panic(__FILE__,"read_write tries to read from "
25112                   "block device NO_DEV", NO_NUM);

```

```

25113         block_size = get_block_size(rip->i_zone[0]);
25114     }
25115
25116     if (!char_spec && !block_spec)
25117         block_size = rip->i_sp->s_block_size;
25118
25119     rdwt_err = OK;           /* set to EIO if disk error occurs */
25120
25121     /* Check for character special files. */
25122     if (char_spec) {
25123         dev_t dev;
25124         dev = (dev_t) rip->i_zone[0];
25125         r = dev_io(op, dev, usr, m_in.buffer, position, m_in.nbytes, oflags);
25126         if (r >= 0) {
25127             cum_io += r;
25128             position += r;
25129             r = OK;
25130         }
25131     } else {
25132         if (rw_flag == WRITING && block_spec == 0) {
25133             /* Check in advance to see if file will grow too big. */
25134             if (position > rip->i_sp->s_max_size - m_in.nbytes)
25135                 return(EFBIG);
25136
25137             /* Check for O_APPEND flag. */
25138             if (oflags & O_APPEND) position = f_size;
25139
25140             /* Clear the zone containing present EOF if hole about
25141                * to be created. This is necessary because all unwritten
25142                * blocks prior to the EOF must read as zeros.
25143                */
25144             if (position > f_size) clear_zone(rip, f_size, 0);
25145         }
25146
25147         /* Pipes are a little different. Check. */
25148         if (rip->i_pipe == I_PIPE) {
25149             r = pipe_check(rip, rw_flag, oflags,
25150                           m_in.nbytes, position, &partial_cnt, 0);
25151             if (r <= 0) return(r);
25152         }
25153
25154         if (partial_cnt > 0) partial_pipe = 1;
25155
25156         /* Split the transfer into chunks that don't span two blocks. */
25157         while (m_in.nbytes != 0) {
25158
25159             off = (unsigned int) (position % block_size); /* offset in blk*/
25160             if (partial_pipe) { /* pipes only */
25161                 chunk = MIN(partial_cnt, block_size - off);
25162             } else
25163                 chunk = MIN(m_in.nbytes, block_size - off);
25164             if (chunk < 0) chunk = block_size - off;
25165
25166             if (rw_flag == READING) {
25167                 bytes_left = f_size - position;
25168                 if (position >= f_size) break; /* we are beyond EOF */
25169                 if (chunk > bytes_left) chunk = (int) bytes_left;
25170             }
25171
25172             /* Read or write 'chunk' bytes. */
25173             r = rw_chunk(rip, position, off, chunk, (unsigned) m_in.nbytes,
25174                         rw_flag, m_in.buffer, seg, usr, block_size, &completed);
25175
25176             if (r != OK) break; /* EOF reached */
25177             if (rdwt_err < 0) break;
25178
25179             /* Update counters and pointers. */

```

```

25180             m_in.buffer += chunk; /* user buffer address */
25181             m_in.nbytes -= chunk; /* bytes yet to be read */
25182             cum_io += chunk; /* bytes read so far */
25183             position += chunk; /* position within the file */
25184
25185             if (partial_pipe) {
25186                 partial_cnt -= chunk;
25187                 if (partial_cnt <= 0) break;
25188             }
25189         }
25190     }
25191
25192     /* On write, update file size and access time. */
25193     if (rw_flag == WRITING) {
25194         if (regular || mode_word == I_DIRECTORY) {
25195             if (position > f_size) rip->i_size = position;
25196         }
25197     } else {
25198         if (rip->i_pipe == I_PIPE) {
25199             if (position >= rip->i_size) {
25200                 /* Reset pipe pointers. */
25201                 rip->i_size = 0; /* no data left */
25202                 position = 0; /* reset reader(s) */
25203                 wf = find_filp(rip, W_BIT);
25204                 if (wf != NIL_FILENO) wf->filp_pos = 0;
25205             }
25206         }
25207     }
25208     f->filp_pos = position;
25209
25210     /* Check to see if read-ahead is called for, and if so, set it up. */
25211     if (rw_flag == READING && rip->i_seek == NO_SEEK && position % block_size == 0
25212         && (regular || mode_word == I_DIRECTORY)) {
25213         rdahed_inode = rip;
25214         rdahedpos = position;
25215     }
25216     rip->i_seek = NO_SEEK;
25217
25218     if (rdwt_err != OK) r = rdwt_err; /* check for disk error */
25219     if (rdwt_err == END_OF_FILE) r = OK;
25220
25221     /* If user-space copying failed, read/write failed. */
25222     if (r == OK && r2 != OK) {
25223         r = r2;
25224     }
25225     if (r == OK) {
25226         if (rw_flag == READING) rip->i_update |= ATIME;
25227         if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
25228         rip->i_dirt = DIRTY; /* inode is thus now dirty */
25229         if (partial_pipe) {
25230             partial_pipe = 0;
25231             /* partial write on pipe with */
25232             /* O_NONBLOCK, return write count */
25233             if (!(oflags & O_NONBLOCK)) {
25234                 fp->fp_cum_io_partial = cum_io;
25235                 suspend(XPIPE); /* partial write on pipe with */
25236                 return(SUSPEND); /* nbytes > PIPE_SIZE - non-atomic */
25237             }
25238         }
25239         fp->fp_cum_io_partial = 0;
25240         return(cum_io);
25241     }
25242     if (bufs_in_use < 0) {
25243         panic(FILE, "end - bufs_in_use negative", bufs_in_use);
25244     }
25245     return(r);
25246 }
```

```

25248 /*=====
25249 *          rw_chunk
25250 *=====
25251 PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff,
25252     seg, usr, block_size, completed)
25253 register struct inode *rip;      /* pointer to inode for file to be rd/wr */
25254 off_t position;               /* position within file to read or write */
25255 unsigned off;                 /* off within the current block */
25256 int chunk;                   /* number of bytes to read or write */
25257 unsigned left;                /* max number of bytes wanted after position */
25258 int rw_flag;                  /* READING or WRITING */
25259 char *buff;                  /* virtual address of the user buffer */
25260 int seg;                     /* T or D segment in user space */
25261 int usr;                     /* which user process */
25262 int block_size;               /* block size of FS operating on */
25263 int *completed;               /* number of bytes copied */
25264 {
25265 /* Read or write (part of) a block. */
25266
25267     register struct buf *bp;
25268     register int r = OK;
25269     int n, block_spec;
25270     block_t b;
25271     dev_t dev;
25272
25273     *completed = 0;
25274
25275     block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
25276     if (block_spec) {
25277         b = position/block_size;
25278         dev = (dev_t) rip->i_zone[0];
25279     } else {
25280         b = read_map(rip, position);
25281         dev = rip->i_dev;
25282     }
25283
25284     if (!block_spec && b == NO_BLOCK) {
25285         if (rw_flag == READING) {
25286             /* Reading from a nonexistent block. Must read as all zeros.*/
25287             bp = get_block(NO_DEV, NO_BLOCK, NORMAL); /* get a buffer */
25288             zero_block(bp);
25289         } else {
25290             /* Writing to a nonexistent block. Create and enter in inode.*/
25291             if ((bp= new_block(rip, position)) == NIL_BUF) return(err_code);
25292         }
25293     } else if (rw_flag == READING) {
25294         /* Read and read ahead if convenient. */
25295         bp = rahead(rip, b, position, left);
25296     } else {
25297         /* Normally an existing block to be partially overwritten is first read
25298          * in. However, a full block need not be read in. If it is already in
25299          * the cache, acquire it, otherwise just acquire a free buffer.
25300
25301         n = (chunk == block_size ? NO_READ : NORMAL);
25302         if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
25303         bp = get_block(dev, b, n);
25304     }
25305
25306     /* In all cases, bp now points to a valid buffer. */
25307     if (bp == NIL_BUF) {
25308         panic(FILE, "bp not valid in rw_chunk, this can't happen", NO_NUM);
25309     }
25310     if (rw_flag == WRITING && chunk != block_size && !block_spec &&
25311         position >= rip->i_size && off == 0) {
25312         zero_block(bp);
25313     }
25314 }
```

```

25315     if (rw_flag == READING) {
25316         /* Copy a chunk from the block buffer to user space. */
25317         r = sys_vircopy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
25318                         usr, seg, (phys_bytes) buff,
25319                         (phys_bytes) chunk);
25320     } else {
25321         /* Copy a chunk from user space to the block buffer. */
25322         r = sys_vircopy(usr, seg, (phys_bytes) buff,
25323                         FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
25324                         (phys_bytes) chunk);
25325         bp->b_dirt = DIRTY;
25326     }
25327     n = (off + chunk == block_size ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
25328     put_block(bp, n);
25329
25330     return(r);
25331 }

25334 /*=====
25335 *          read_map
25336 *=====
25337 PUBLIC block_t read_map(rip, position)
25338 register struct inode *rip;      /* ptr to inode to map from */
25339 off_t position;                /* position in file whose blk wanted */
25340 {
25341 /* Given an inode and a position within the corresponding file, locate the
25342 * block (not zone) number in which that position is to be found and return it.
25343 */
25344
25345 register struct buf *bp;
25346 register zone_t z;
25347 int scale, boff, dzones, nr_indirects, index, zind, ex;
25348 block_t b;
25349 long excess, zone, block_pos;
25350
25351 scale = rip->i_sp->s_log_zone_size; /* for block-zone conversion */
25352 block_pos = position/rip->i_sp->s_block_size; /* relative blk # in file */
25353 zone = block_pos >> scale; /* position's zone */
25354 boff = (int) (block_pos - (zone << scale)); /* relative blk # within zone */
25355 dzones = rip->i_ndzones;
25356 nr_indirects = rip->i_nindirs;
25357
25358 /* Is 'position' to be found in the inode itself? */
25359 if (zone < dzones) {
25360     zind = (int) zone; /* index should be an int */
25361     z = rip->i_zone[zind];
25362     if (z == NO_ZONE) return(NO_BLOCK);
25363     b = ((block_t) z << scale) + boff;
25364     return(b);
25365 }
25366
25367 /* It is not in the inode, so it must be single or double indirect. */
25368 excess = zone - dzones; /* first Vx_NR_DZONES don't count */
25369
25370 if (excess < nr_indirects) {
25371     /* 'position' can be located via the single indirect block. */
25372     z = rip->i_zone[dzones];
25373 } else {
25374     /* 'position' can be located via the double indirect block. */
25375     if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
25376     excess -= nr_indirects; /* single indir doesn't count */
25377     b = (block_t) z << scale;
25378     bp = get_block(rip->i_dev, b, NORMAL); /* get double indirect block */
25379     index = (int) (excess/nr_indirects);
25380     z = rd_indir(bp, index); /* z= zone for single*/
25381     put_block(bp, INDIRECT_BLOCK); /* release double ind block */

```

```

25382         excess = excess % nr_indirects;           /* index into single ind blk */
25383     }
25384     .
25385     /* 'z' is zone num for single indirect block; 'excess' is index into it. */
25386     if (z == NO_ZONE) return(NO_BLOCK);
25387     b = (block_t) z << scale;                   /* b is blk # for single ind */
25388     bp = get_block(rip->i_dev, b, NORMAL);      /* get single indirect block */
25389     ex = (int) excess;                          /* need an integer */
25390     z = rd.indir(bp, ex);                      /* get block pointed to */
25391     put_block(bp, INDIRECT_BLOCK);              /* release single indir blk */
25392     if (z == NO_ZONE) return(NO_BLOCK);
25393     b = ((block_t) z << scale) + boff;
25394     return(b);
25395 }

25396 /*=====
25397 *                         rd_indir
25398 *=====
25399 */
25400 PUBLIC zone_t rd_indir(bp, index)
25401 struct buf *bp;          /* pointer to indirect block */
25402 int index;               /* index into *bp */
25403 {
25404 /* Given a pointer to an indirect block, read one entry. The reason for
25405 * making a separate routine out of this is that there are four cases:
25406 * V1 (IBM and 68000), and V2 (IBM and 68000).
25407 */
25408
25409     struct super_block *sp;
25410     zone_t zone;                /* V2 zones are longs (shorts in V1) */
25411
25412     sp = get_super(bp->b_dev);   /* need super block to find file sys type */
25413
25414 /* read a zone from an indirect block */
25415 if (sp->s_version == V1)
25416     zone = (zone_t) conv2(sp->s_native, (int) bp->b_v1_ind[index]);
25417 else
25418     zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);
25419
25420 if (zone != NO_ZONE &&
25421     (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
25422     printf("Illegal zone number %d in indirect block, index %d\n",
25423           (long) zone, index);
25424     panic(__FILE__, "check file system", NO_NUM);
25425 }
25426 return(zone);
25427 }

25428 /*=====
25429 *                         read_ahead
25430 *=====
25431 */
25432 PUBLIC void read_ahead()
25433 {
25434 /* Read a block into the cache before it is needed. */
25435     int block_size;
25436     register struct inode *rip;
25437     struct buf *bp;
25438     block_t b;
25439
25440     rip = rdahed_inode;           /* pointer to inode to read ahead from */
25441     block_size = get_block_size(rip->i_dev);
25442     rdahed_inode = NIL_INODE;    /* turn off read ahead */
25443     if ((b = read_map(rip, rdahedpos)) == NO_BLOCK) return;        /* at EOF */
25444     bp = rahead(rip, b, rdahedpos, block_size);
25445     put_block(bp, PARTIAL_DATA_BLOCK);
25446 }

```

```

25448 /*=====
25449 *          rahead
25450 *=====
25451 PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
25452 register struct inode *rip;      /* pointer to inode for file to be read */
25453 block_t baseblock;            /* block at current position */
25454 off_t position;              /* position within file */
25455 unsigned bytes_ahead;        /* bytes beyond position for immediate use */
25456 {
25457 /* Fetch a block from the cache or the device. If a physical read is
25458 * required, prefetch as many more blocks as convenient into the cache.
25459 * This usually covers bytes_ahead and is at least BLOCKS_MINIMUM.
25460 * The device driver may decide it knows better and stop reading at a
25461 * cylinder boundary (or after an error). Rw_scattered() puts an optional
25462 * flag on all reads to allow this.
25463 */
25464     int block_size;
25465 /* Minimum number of blocks to prefetch. */
25466 # define BLOCKS_MINIMUM      (NR_BUFS < 50 ? 18 : 32)
25467     int block_spec, scale, read_q_size;
25468     unsigned int blocks_ahead, fragment;
25469     block_t block, blocks_left;
25470     off_t ind1_pos;
25471     dev_t dev;
25472     struct buf *bp;
25473     static struct buf *read_q[NR_BUFS];
25474
25475     block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
25476     if (block_spec) {
25477         dev = (dev_t) rip->i_zone[0];
25478     } else {
25479         dev = rip->i_dev;
25480     }
25481     block_size = get_block_size(dev);
25482
25483     block = baseblock;
25484     bp = get_block(dev, block, PREFETCH);
25485     if (bp->b_dev != NO_DEV) return(bp);
25486
25487 /* The best guess for the number of blocks to prefetch: A lot.
25488 * It is impossible to tell what the device looks like, so we don't even
25489 * try to guess the geometry, but leave it to the driver.
25490 */
25491 /* The floppy driver can read a full track with no rotational delay, and it
25492 * avoids reading partial tracks if it can, so handing it enough buffers to
25493 * read two tracks is perfect. (Two, because some diskette types have
25494 * an odd number of sectors per track, so a block may span tracks.)
25495 */
25496 /* The disk drivers don't try to be smart. With todays disks it is
25497 * impossible to tell what the real geometry looks like, so it is best to
25498 * read as much as you can. With luck the caching on the drive allows
25499 * for a little time to start the next read.
25500 */
25501 /* The current solution below is a bit of a hack, it just reads blocks from
25502 * the current file position hoping that more of the file can be found. A
25503 * better solution must look at the already available zone pointers and
25504 * indirect blocks (but don't call read_map!).
25505 */
25506
25507     fragment = position % block_size;
25508     position -= fragment;
25509     bytes_ahead += fragment;
25510
25511     blocks_ahead = (bytes_ahead + block_size - 1) / block_size;
25512
25513     if (block_spec && rip->i_size == 0) {
25514         blocks_left = NR_IOREQS;

```

```

25515 } else {
25516     blocks_left = (rip->i_size - position + block_size - 1) / block_size;
25517
25518     /* Go for the first indirect block if we are in its neighborhood. */
25519     if (!block_spec) {
25520         scale = rip->i_sp->s_log_zone_size;
25521         ind1_pos = (off_t) rip->i_ndzones * (block_size << scale);
25522         if (position <= ind1_pos && rip->i_size > ind1_pos) {
25523             blocks_ahead++;
25524             blocks_left++;
25525         }
25526     }
25527 }
25528
25529 /* No more than the maximum request. */
25530 if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;
25531
25532 /* Read at least the minimum number of blocks, but not after a seek. */
25533 if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
25534     blocks_ahead = BLOCKS_MINIMUM;
25535
25536 /* Can't go past end of file. */
25537 if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;
25538
25539 read_q_size = 0;
25540
25541 /* Acquire block buffers. */
25542 for (;;) {
25543     read_q[read_q_size++] = bp;
25544
25545     if (--blocks_ahead == 0) break;
25546
25547     /* Don't trash the cache, leave 4 free. */
25548     if (bufs_in_use >= NR_BUFS - 4) break;
25549
25550     block++;
25551
25552     bp = get_block(dev, block, PREFETCH);
25553     if (bp->b_dev != NO_DEV) {
25554         /* Oops, block already in the cache, get out. */
25555         put_block(bp, FULL_DATA_BLOCK);
25556         break;
25557     }
25558 }
25559 rw_scattered(dev, read_q, read_q_size, READING);
25560 return(get_block(dev, baseblock, NORMAL));
25561 }

```

```
+++++
servers/fs/write.c
+++++
```

```

25600 /* This file is the counterpart of "read.c". It contains the code for writing
25601 * insofar as this is not contained in read_write().
25602 *
25603 * The entry points into this file are
25604 *   do_write:    call read_write to perform the WRITE system call
25605 *   clear_zone:  erase a zone in the middle of a file
25606 *   new_block:   acquire a new block
25607 */
25608
25609 #include "fs.h"
25610 #include <string.h>
25611 #include "buf.h"

```

```

25612 #include "file.h"
25613 #include "fproc.h"
25614 #include "inode.h"
25615 #include "super.h"
25616
25617 FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position,
25618                                     zone_t new_zone)
25619                                     );
25620 FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );
25621
25622 /*=====
25623 *          do_write
25624 *=====
25625 PUBLIC int do_write()
25626 {
25627 /* Perform the write(fd, buffer, nbytes) system call. */
25628     return(read_write(WRITING));
25629 }
25630
25631 /*=====
25632 *          write_map
25633 *=====
25634 PRIVATE int write_map(rip, position, new_zone)
25635 register struct inode *rip;      /* pointer to inode to be changed */
25636 off_t position;               /* file address to be mapped */
25637 zone_t new_zone;              /* zone # to be inserted */
25638 {
25639 /* Write a new zone into an inode. */
25640     int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
25641     zone_t z, z1;
25642     register block_t b;
25643     long excess, zone;
25644     struct buf *bp;
25645
25646     rip->i_dirt = DIRTY;           /* inode will be changed */
25647     bp = NIL_BUF;
25648     scale = rip->i_sp->s_log_zone_size;        /* for zone-block conversion */
25649     /* relative zone # to insert */
25650     zone = (position/rip->i_sp->s_block_size) >> scale;
25651     zones = rip->i_ndzones;           /* # direct zones in the inode */
25652     nr_indirects = rip->i_nindirs;/* # indirect zones per indirect block */
25653
25654 /* Is 'position' to be found in the inode itself? */
25655 if (zone < zones) {
25656     zindex = (int) zone;    /* we need an integer here */
25657     rip->i_zone[zindex] = new_zone;
25658     return(OK);
25659 }
25660
25661 /* It is not in the inode, so it must be single or double indirect. */
25662 excess = zone - zones;           /* first Vx_NR_DZONES don't count */
25663 new_ind = FALSE;
25664 new_dbl = FALSE;
25665
25666 if (excess < nr_indirects) {
25667     /* 'position' can be located via the single indirect block. */
25668     z1 = rip->i_zone[zones];           /* single indirect zone */
25669     single = TRUE;
25670 } else {
25671     /* 'position' can be located via the double indirect block. */
25672     if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
25673         /* Create the double indirect block. */
25674         if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
25675             return(err_code);
25676         rip->i_zone[zones+1] = z;
25677         new_dbl = TRUE; /* set flag for later */
25678     }
}

```



```

25747 PUBLIC void clear_zone(rip, pos, flag)
25748 register struct inode *rip;      /* inode to clear */
25749 off_t pos;                  /* points to block to clear */
25750 int flag;                  /* 0 if called by read_write, 1 by new_block */
25751 {
25752 /* Zero a zone, possibly starting in the middle. The parameter 'pos' gives
25753 * a byte in the first block to be zeroed. Clearzone() is called from
25754 * read_write and new_block().
25755 */
25756
25757     register struct buf *bp;
25758     register block_t b, blo, bhi;
25759     register off_t next;
25760     register int scale;
25761     register zone_t zone_size;
25762
25763 /* If the block size and zone size are the same, clear_zone() not needed. */
25764 scale = rip->i_sp->s_log_zone_size;
25765 if (scale == 0) return;
25766
25767 zone_size = (zone_t) rip->i_sp->s_block_size << scale;
25768 if (flag == 1) pos = (pos/zone_size) * zone_size;
25769 next = pos + rip->i_sp->s_block_size - 1;
25770
25771 /* If 'pos' is in the last block of a zone, do not clear the zone. */
25772 if (next/zone_size != pos/zone_size) return;
25773 if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
25774 bhi = ( ((blo>>scale)+1) << scale) - 1;
25775
25776 /* Clear all the blocks between 'blo' and 'bhi'. */
25777 for (b = blo; b <= bhi; b++) {
25778     bp = get_block(rip->i_dev, b, NO_READ);
25779     zero_block(bp);
25780     put_block(bp, FULL_DATA_BLOCK);
25781 }
25782 }

25784 /*=====
25785 *          new_block
25786 *=====
25787 */
25788 PUBLIC struct buf *new_block(rip, position)
25789 register struct inode *rip;      /* pointer to inode */
25790 off_t position;                /* file pointer */
25791 {
25792 /* Acquire a new block and return a pointer to it. Doing so may require
25793 * allocating a complete zone, and then returning the initial block.
25794 * On the other hand, the current zone may still have some unused blocks.
25795 */
25796     register struct buf *bp;
25797     block_t b, base_block;
25798     zone_t z;
25799     zone_t zone_size;
25800     int scale, r;
25801     struct super_block *sp;
25802
25803 /* Is another block available in the current zone? */
25804 if ( (b = read_map(rip, position)) == NO_BLOCK) {
25805     /* Choose first zone if possible. */
25806     /* Lose if the file is nonempty but the first zone number is NO_ZONE
25807     * corresponding to a zone full of zeros. It would be better to
25808     * search near the last real zone.
25809     */
25810     if (rip->i_zone[0] == NO_ZONE) {
25811         sp = rip->i_sp;
25812         z = sp->s_firstdatazone;
25813     } else {
25814         z = rip->i_zone[0];      /* hunt near first zone */

```

```

25815      }
25816      if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
25817      if ( (r = write_map(rip, position, z)) != OK) {
25818          free_zone(rip->i_dev, z);
25819          err_code = r;
25820          return(NIL_BUF);
25821      }
25822
25823      /* If we are not writing at EOF, clear the zone, just to be safe. */
25824      if ( position != rip->i_size) clear_zone(rip, position, 1);
25825      scale = rip->i_sp->s_log_zone_size;
25826      base_block = (block_t) z << scale;
25827      zone_size = (zone_t) rip->i_sp->s_block_size << scale;
25828      b = base_block + (block_t)((position % zone_size)/rip->i_sp->s_block_size);
25829  }
25830
25831  bp = get_block(rip->i_dev, b, NO_READ);
25832  zero_block(bp);
25833  return(bp);
25834 }

25836 /*=====
25837 *          zero_block
25838 *=====
25839 PUBLIC void zero_block(bp)
25840 register struct buf *bp;      /* pointer to buffer to zero */
25841 {
25842 /* Zero a block. */
25843     memset(bp->b_data, 0, MAX_BLOCK_SIZE);
25844     bp->b_dirt = DIRTY;
25845 }

+++++
servers/fs/pipe.c
+++++

25900 /* This file deals with the suspension and revival of processes. A process can
25901 * be suspended because it wants to read or write from a pipe and can't, or
25902 * because it wants to read or write from a special file and can't. When a
25903 * process can't continue it is suspended, and revived later when it is able
25904 * to continue.
25905 *
25906 * The entry points into this file are
25907 *   do_pipe:    perform the PIPE system call
25908 *   pipe_check: check to see that a read or write on a pipe is feasible now
25909 *   suspend:    suspend a process that cannot do a requested read or write
25910 *   release:    check to see if a suspended process can be released and do
25911 *                 it
25912 *   revive:    mark a suspended process as able to run again
25913 *   do_unpause: a signal has been sent to a process; see if it suspended
25914 */
25915
25916 #include "fs.h"
25917 #include <fcntl.h>
25918 #include <signal.h>
25919 #include <minix/callnr.h>
25920 #include <minix/com.h>
25921 #include <sys/select.h>
25922 #include <sys/time.h>
25923 #include "file.h"
25924 #include "fproc.h"
25925 #include "inode.h"
25926 #include "param.h"
25927 #include "super.h"
25928 #include "select.h"

```

```

25929
25930 /*=====
25931 *          do_pipe
25932 *=====
25933 PUBLIC int do_pipe()
25934 {
25935 /* Perform the pipe(fil_des) system call. */
25936
25937     register struct fproc *rfp;
25938     register struct inode *rip;
25939     int r;
25940     struct filp *fil_ptr0, *fil_ptr1;
25941     int fil_des[2];           /* reply goes here */
25942
25943 /* Acquire two file descriptors. */
25944     rfp = fp;
25945     if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);
25946     rfp->fp_filp[fil_des[0]] = fil_ptr0;
25947     fil_ptr0->filp_count = 1;
25948     if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK) {
25949         rfp->fp_filp[fil_des[0]] = NIL_FILP;
25950         fil_ptr0->filp_count = 0;
25951         return(r);
25952     }
25953     rfp->fp_filp[fil_des[1]] = fil_ptr1;
25954     fil_ptr1->filp_count = 1;
25955
25956 /* Make the inode on the pipe device. */
25957     if ( (rip = alloc_inode(root_dev, I_REGULAR) ) == NIL_INODE) {
25958         rfp->fp_filp[fil_des[0]] = NIL_FILP;
25959         fil_ptr0->filp_count = 0;
25960         rfp->fp_filp[fil_des[1]] = NIL_FILP;
25961         fil_ptr1->filp_count = 0;
25962         return(err_code);
25963     }
25964
25965     if (read_only(rip) != OK)
25966         panic(__FILE__,"pipe device is read only", NO_NUM);
25967
25968     rip->i_pipe = I_PIPE;
25969     rip->i_mode &= ~I_REGULAR;
25970     rip->i_mode |= I_NAMED_PIPE; /* pipes and FIFOs have this bit set */
25971     fil_ptr0->filp_ino = rip;
25972     fil_ptr0->filp_flags = O_RDONLY;
25973     dup_inode(rip);           /* for double usage */
25974     fil_ptr1->filp_ino = rip;
25975     fil_ptr1->filp_flags = O_WRONLY;
25976     rw_inode(rip, WRITING); /* mark inode as allocated */
25977     m_out.reply_i1 = fil_des[0];
25978     m_out.reply_i2 = fil_des[1];
25979     rip->i_update = ATIME | CTIME | MTIME;
25980     return(OK);
25981 }

25982 /*=====
25983 *          pipe_check
25984 *=====
25985 */
25986 PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite, notouch)
25987 register struct inode *rip;      /* the inode of the pipe */
25988 int rw_flag;                  /* READING or WRITING */
25989 int oflags;                   /* flags set by open or fcntl */
25990 register int bytes;           /* bytes to be read or written (all chunks) */
25991 register off_t position;      /* current file position */
25992 int *canwrite;                /* return: number of bytes we can write */
25993 int notouch;                 /* check only */
25994 {
25995 /* Pipes are a little different. If a process reads from an empty pipe for

```

```

25996 * which a writer still exists, suspend the reader. If the pipe is empty
25997 * and there is no writer, return 0 bytes. If a process is writing to a
25998 * pipe and no one is reading from it, give a broken pipe error.
25999 */
26000
26001 /* If reading, check for empty pipe. */
26002 if (rw_flag == READING) {
26003     if (position >= rip->i_size) {
26004         /* Process is reading from an empty pipe. */
26005         int r = 0;
26006         if (find_filp(rip, W_BIT) != NIL_FILP) {
26007             /* Writer exists */
26008             if (oflags & O_NONBLOCK) {
26009                 r = EAGAIN;
26010             } else {
26011                 if (!notouch)
26012                     suspend(XPIPE); /* block reader */
26013                 r = SUSPEND;
26014             }
26015             /* If need be, activate sleeping writers. */
26016             if (susp_count > 0 && !notouch)
26017                 release(rip, WRITE, susp_count);
26018         }
26019         return(r);
26020     }
26021 } else {
26022     /* Process is writing to a pipe. */
26023     if (find_filp(rip, R_BIT) == NIL_FILP) {
26024         /* Tell kernel to generate a SIGPIPE signal. */
26025         if (!notouch)
26026             sys_kill((int)(fp - fproc), SIGPIPE);
26027         return(EPIPE);
26028     }
26029
26030     if (position + bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
26031         if ((oflags & O_NONBLOCK)
26032             && bytes < PIPE_SIZE(rip->i_sp->s_block_size))
26033             return(EAGAIN);
26034         else if ((oflags & O_NONBLOCK)
26035             && bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
26036             if ((*canwrite = (PIPE_SIZE(rip->i_sp->s_block_size)
26037                 - position)) > 0) {
26038                 /* Do a partial write. Need to wakeup reader */
26039                 if (!notouch)
26040                     release(rip, READ, susp_count);
26041                 return(1);
26042             } else {
26043                 return(EAGAIN);
26044             }
26045         }
26046         if (bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
26047             if ((*canwrite = PIPE_SIZE(rip->i_sp->s_block_size)
26048                 - position) > 0) {
26049                 /* Do a partial write. Need to wakeup reader
26050                 * since we'll suspend ourselves in read_write()
26051                 */
26052                 release(rip, READ, susp_count);
26053                 return(1);
26054             }
26055         }
26056         if (!notouch)
26057             suspend(XPIPE); /* stop writer -- pipe full */
26058         return(SUSPEND);
26059     }
26060
26061     /* Writing to an empty pipe. Search for suspended reader. */
26062     if (position == 0 && !notouch)

```

```

26063             release(rip, READ, susp_count);
26064     }
26065
26066     *canwrite = 0;
26067     return(1);
26068 }

26070 /*=====
26071 *          suspend
26072 *=====
26073 PUBLIC void suspend(task)
26074 int task;           /* who is proc waiting for? (PIPE = pipe) */
26075 {
26076 /* Take measures to suspend the processing of the present system call.
26077 * Store the parameters to be used upon resuming in the process table.
26078 * (Actually they are not used when a process is waiting for an I/O device,
26079 * but they are needed for pipes, and it is not worth making the distinction.)
26080 * The SUSPEND pseudo error should be returned after calling suspend().
26081 */
26082
26083 if (task == XPIPE || task == XOPEN) susp_count++; /* #procs susp'ed on pipe*/
26084 fp->fp_suspended = SUSPENDED;
26085 fp->fp_fd = m_in.fd << 8 | call_nr;
26086 fp->fp_task = -task;
26087 if (task == XLOCK) {
26088     fp->fp_buffer = (char *) m_in.name1;    /* third arg to fcntl() */
26089     fp->fp_nbytes = m_in.request;           /* second arg to fcntl() */
26090 } else {
26091     fp->fp_buffer = m_in.buffer;            /* for reads and writes */
26092     fp->fp_nbytes = m_in.nbytes;
26093 }
26094 }

26095 /*=====
26096 *          release
26097 *=====
26098 */
26099 PUBLIC void release(ip, call_nr, count)
26100 register struct inode *ip;      /* inode of pipe */
26101 int call_nr;                 /* READ, WRITE, OPEN or CREAT */
26102 int count;                   /* max number of processes to release */
26103 {
26104 /* Check to see if any process is hanging on the pipe whose inode is in 'ip'.
26105 * If one is, and it was trying to perform the call indicated by 'call_nr',
26106 * release it.
26107 */
26108
26109 register struct fproc *rp;
26110 struct filp *f;
26111
26112 /* Trying to perform the call also includes SELECTING on it with that
26113 * operation.
26114 */
26115 if (call_nr == READ || call_nr == WRITE) {
26116     int op;
26117     if (call_nr == READ)
26118         op = SEL_RD;
26119     else
26120         op = SEL_WR;
26121     for(f = &filp[0]; f < &filp[NR_FILENO]; f++) {
26122         if (f->filp_count < 1 || !(f->filp_pipe_select_ops & op) ||
26123             f->filp_ino != ip)
26124             continue;
26125         select_callback(f, op);
26126         f->filp_pipe_select_ops &= ~op;
26127     }
26128 }
26129 }
```

```

26130 /* Search the proc table. */
26131 for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++) {
26132     if (rp->fp_suspended == SUSPENDED &&
26133         rp->fp_revived == NOT_REVIVING &&
26134         (rp->fp_fd & BYTE) == call_nr &&
26135         rp->fp_filp[rp->fp_fd>8]->filp_ino == ip) {
26136         revive((int)(rp - fproc), 0);
26137         susp_count--; /* keep track of who is suspended */
26138         if (--count == 0) return;
26139     }
26140 }
26141 }

26143 /*=====
26144 *          revive
26145 *=====
26146 PUBLIC void revive(proc_nr, returned)
26147 int proc_nr;           /* process to revive */
26148 int returned;          /* if hanging on task, how many bytes read */
26149 {
26150 /* Revive a previously blocked process. When a process hangs on tty, this
26151 * is the way it is eventually released.
26152 */
26153
26154 register struct fproc *rfp;
26155 register int task;
26156
26157 if (proc_nr < 0 || proc_nr >= NR_PROCS)
26158     panic(__FILE__, "revive err", proc_nr);
26159 rfp = &fproc[proc_nr];
26160 if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING) return;
26161
26162 /* The 'reviving' flag only applies to pipes. Processes waiting for TTY get
26163 * a message right away. The revival process is different for TTY and pipes.
26164 * For select and TTY revival, the work is already done, for pipes it is not:
26165 * the proc must be restarted so it can try again.
26166 */
26167 task = -rfp->fp_task;
26168 if (task == XPIPE || task == XLOCK) {
26169     /* Revive a process suspended on a pipe or lock. */
26170     rfp->fp_revived = REVIVING;
26171     reviving++; /* process was waiting on pipe or lock */
26172 } else {
26173     rfp->fp_suspended = NOT_SUSPENDED;
26174     if (task == XOPEN) /* process blocked in open or create */
26175         reply(proc_nr, rfp->fp_fd>8);
26176     else if (task == XSELECT) {
26177         reply(proc_nr, returned);
26178     } else {
26179         /* Revive a process suspended on TTY or other device. */
26180         rfp->fp_nbytes = returned; /*pretend it wants only what there is*/
26181         reply(proc_nr, returned); /* unblock the process */
26182     }
26183 }
26184 }

26186 /*=====
26187 *          do_unpause
26188 *=====
26189 PUBLIC int do_unpause()
26190 {
26191 /* A signal has been sent to a user who is paused on the file system.
26192 * Abort the system call with the EINTR error message.
26193 */
26194

```

```

26195 register struct fproc *rfp;
26196 int proc_nr, task, fild;
26197 struct filp *f;
26198 dev_t dev;
26199 message mess;
26200
26201 if (who > PM_PROC_NR) return(EPERM);
26202 proc_nr = m_in.pro;
26203 if (proc_nr < 0 || proc_nr >= NR_PROCS)
26204     panic(FILE, "unpause err 1", proc_nr);
26205 rfp = &fproc[proc_nr];
26206 if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
26207 task = -rfp->fp_task;
26208
26209 switch (task) {
26210     case XPIPE:           /* process trying to read or write a pipe */
26211         break;
26212
26213     case XLOCK:           /* process trying to set a lock with Fcntl */
26214         break;
26215
26216     case XSELECT:          /* process blocking on select() */
26217         select_forget(proc_nr);
26218         break;
26219
26220     case XOPEN:            /* process trying to open a fifo */
26221         break;
26222
26223     default:               /* process trying to do device I/O (e.g. tty) */
26224         fild = (rfp->fp_fd >> 8) & BYTE; /* extract file descriptor */
26225         if (fld < 0 || fld >= OPEN_MAX)
26226             panic(FILE, "unpause err 2", NO_NUM);
26227         f = rfp->fp_filp[fild];
26228         dev = (dev_t) f->filp_ino->i_zone[0]; /* device hung on */
26229         mess.TTY_LINE = (dev >> MINOR) & BYTE;
26230         mess.PROC_NR = proc_nr;
26231
26232         /* Tell kernel R or W. Mode is from current call, not open. */
26233         mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
26234         mess.m_type = CANCEL;
26235         fp = rfp; /* hack - cttty_io uses fp */
26236         (*dmap[(dev >> MAJOR) & BYTE].dmap_io)(task, &mess);
26237     }
26238
26239     rfp->fp_suspended = NOT_SUSPENDED;
26240     reply(proc_nr, EINTR); /* signal interrupted call */
26241     return(OK);
26242 }

26244 /*=====
26245 *           select_request_pipe
26246 *=====
26247 PUBLIC int select_request_pipe(struct filp *f, int *ops, int block)
26248 {
26249     int orig_ops, r = 0, err, canwrite;
26250     orig_ops = *ops;
26251     if ((*ops & SEL_RD)) {
26252         if ((err = pipe_check(f->filp_ino, READING, 0,
26253                             1, f->filp_pos, &canwrite, 1)) != SUSPEND)
26254             r |= SEL_RD;
26255         if (err < 0 && err != SUSPEND && (*ops & SEL_ERR))
26256             r |= SEL_ERR;
26257     }
26258     if ((*ops & SEL_WR)) {
26259         if ((err = pipe_check(f->filp_ino, WRITING, 0,
26260                             1, f->filp_pos, &canwrite, 1)) != SUSPEND)
26261             r |= SEL_WR;

```

```

26262             if (err < 0 && err != SUSPEND && (*ops & SEL_ERR))
26263                 r |= SEL_ERR;
26264         }
26265
26266         *ops = r;
26267
26268         if (!r && block) {
26269             f->filp_pipe_select_ops |= orig_ops;
26270         }
26271
26272         return SEL_OK;
26273     }

26275 /*=====
26276 *          select_match_pipe
26277 *=====
26278 PUBLIC int select_match_pipe(struct filp *f)
26279 {
26280     /* recognize either pipe or named pipe (FIFO) */
26281     if (f && f->filp_ino && (f->filp_ino->i_mode & I_NAMED_PIPE))
26282         return 1;
26283     return 0;
26284 }

+++++
servers/fs/path.c
+++++

26300 /* This file contains the procedures that look up path names in the directory
26301 * system and determine the inode number that goes with a given path name.
26302 *
26303 * The entry points into this file are
26304 * eat_path: the 'main' routine of the path-to-inode conversion mechanism
26305 * last_dir: find the final directory on a given path
26306 * advance: parse one component of a path name
26307 * search_dir: search a directory for a string and return its inode number
26308 */
26309
26310 #include "fs.h"
26311 #include <string.h>
26312 #include <minix/callnr.h>
26313 #include "buf.h"
26314 #include "file.h"
26315 #include "fproc.h"
26316 #include "inode.h"
26317 #include "super.h"
26318
26319 PUBLIC char dot1[2] = ".";
26320 PUBLIC char dot2[3] = "..";
26321 FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );
26322
26323 /*=====
26324 *          eat_path
26325 *=====
26326 PUBLIC struct inode *eat_path(path)
26327 char *path; /* the path name to be parsed */
26328 {
26329     /* Parse the path 'path' and put its inode in the inode table. If not possible,
26330     * return NIL_INODE as function value and an error code in 'err_code'.
26331     */
26332
26333

```

```

26334 register struct inode *ldip, *rip;
26335 char string[NAME_MAX];           /* hold 1 path component name here */
26336
26337 /* First open the path down to the final directory. */
26338 if ( (ldip = last_dir(path, string)) == NIL_INODE) {
26339     return(NIL_INODE);           /* we couldn't open final directory */
26340 }
26341
26342 /* The path consisting only of "/" is a special case, check for it. */
26343 if (string[0] == '\0') return(1dip);
26344
26345 /* Get final component of the path. */
26346 rip = advance(1dip, string);
26347 put_inode(1dip);
26348 return(rip);
26349 }

26351 /*=====
26352 *                         last_dir
26353 *=====
26354 PUBLIC struct inode *last_dir(path, string)
26355 char *path;                  /* the path name to be parsed */
26356 char string[NAME_MAX];       /* the final component is returned here */
26357 {
26358 /* Given a path, 'path', located in the fs address space, parse it as
26359 * far as the last directory, fetch the inode for the last directory into
26360 * the inode table, and return a pointer to the inode. In
26361 * addition, return the final component of the path in 'string'.
26362 * If the last directory can't be opened, return NIL_INODE and
26363 * the reason for failure in 'err_code'.
26364 */
26365
26366 register struct inode *rip;
26367 register char *new_name;
26368 register struct inode *new_ip;
26369
26370 /* Is the path absolute or relative? Initialize 'rip' accordingly. */
26371 rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);
26372
26373 /* If dir has been removed or path is empty, return ENOENT. */
26374 if (rip->i_nlinks == 0 || *path == '\0') {
26375     err_code = ENOENT;
26376     return(NIL_INODE);
26377 }
26378
26379 dup_inode(rip);             /* inode will be returned with put_inode */
26380
26381 /* Scan the path component by component. */
26382 while (TRUE) {
26383     /* Extract one component. */
26384     if ( (new_name = get_name(path, string)) == (char*) 0) {
26385         put_inode(rip); /* bad path in user space */
26386         return(NIL_INODE);
26387     }
26388     if (*new_name == '\0') {
26389         if ( (rip->i_mode & I_TYPE) == I_DIRECTORY) {
26390             return(rip); /* normal exit */
26391         } else {
26392             /* last file of path prefix is not a directory */
26393             put_inode(rip);
26394             err_code = ENOTDIR;
26395             return(NIL_INODE);
26396         }
26397     }
26398
26399     /* There is more path. Keep parsing. */
26400     new_ip = advance(rip, string);
26401     put_inode(rip);           /* rip either obsolete or irrelevant */

```

```

26402         if (new_ip == NIL_INODE) return(NIL_INODE);
26403
26404     /* The call to advance() succeeded. Fetch next component. */
26405     path = new_name;
26406     rip = new_ip;
26407 }
26408 }

26410 /*=====
26411 *          get_name
26412 *=====
26413 PRIVATE char *get_name(old_name, string)
26414 char *old_name;           /* path name to parse */
26415 char string[NAME_MAX];    /* component extracted from 'old_name' */
26416 {
26417 /* Given a pointer to a path name in fs space, 'old_name', copy the next
26418 * component to 'string' and pad with zeros. A pointer to that part of
26419 * the name as yet unparsed is returned. Roughly speaking,
26420 * 'get_name' = 'old_name' - 'string'.
26421 *
26422 * This routine follows the standard convention that /usr/ast, /usr//ast,
26423 * //usr///ast and /usr/ast/ are all equivalent.
26424 */
26425
26426 register int c;
26427 register char *np, *rnp;
26428
26429 np = string;           /* 'np' points to current position */
26430 rnp = old_name;        /* 'rnp' points to unparsed string */
26431 while ( (c = *rnp) == '/') rnp++; /* skip leading slashes */
26432
26433 /* Copy the unparsed path, 'old_name', to the array, 'string'. */
26434 while ( rnp < &old_name[PATH_MAX] && c != '/' && c != '\0') {
26435     if (np < &string[NAME_MAX]) *np++ = c;
26436     c = *++rnp;           /* advance to next character */
26437 }
26438
26439 /* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
26440 while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;
26441
26442 if (np < &string[NAME_MAX]) *np = '\0';      /* Terminate string */
26443
26444 if (rnp >= &old_name[PATH_MAX]) {
26445     err_code = ENAMETOOLONG;
26446     return((char *) 0);
26447 }
26448 return(rnp);
26449 }

26450 /*=====
26451 *          advance
26452 *=====
26453 PUBLIC struct inode *advance(dirp, string)
26454 struct inode *dirp;           /* inode for directory to be searched */
26455 char string[NAME_MAX];       /* component name to look for */
26456 {
26457 /* Given a directory and a component of a path, look up the component in
26458 * the directory, find the inode, open it, and return a pointer to its inode
26459 * slot. If it can't be done, return NIL_INODE.
26460 */
26461
26462 register struct inode *rip;
26463 struct inode *rip2;
26464 register struct super_block *sp;
26465 int r, inumb;
26466 dev_t mnt_dev;
26467 ino_t numb;

```

```

26469
26470 /* If 'string' is empty, yield same inode straight away. */
26471 if (string[0] == '\0') { return(get_inode(dirp->i_dev, (int) dirp->i_num)); }
26472
26473 /* Check for NIL_INODE. */
26474 if (dirp == NIL_INODE) { return(NIL_INODE); }
26475
26476 /* If 'string' is not present in the directory, signal error. */
26477 if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
26478     err_code = r;
26479     return(NIL_INODE);
26480 }
26481
26482 /* Don't go beyond the current root directory, unless the string is dot2. */
26483 if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
26484     return(get_inode(dirp->i_dev, (int) dirp->i_num));
26485
26486 /* The component has been found in the directory. Get inode. */
26487 if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE) {
26488     return(NIL_INODE);
26489 }
26490
26491 if (rip->i_num == ROOT_INODE) {
26492     if (dirp->i_num == ROOT_INODE) {
26493         if (string[1] == ') {
26494             for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
26495                 if (sp->s_dev == rip->i_dev) {
26496                     /* Release the root inode. Replace by the
26497                     * inode mounted on.
26498                     */
26499                     put_inode(rip);
26500                     mnt_dev = sp->s_imount->i_dev;
26501                     inumb = (int) sp->s_imount->i_num;
26502                     rip2 = get_inode(mnt_dev, inumb);
26503                     rip = advance(rip2, string);
26504                     put_inode(rip2);
26505                     break;
26506                 }
26507             }
26508         }
26509     }
26510     if (rip == NIL_INODE) return(NIL_INODE);
26511
26512 /* See if the inode is mounted on. If so, switch to root directory of the
26513 * mounted file system. The super_block provides the linkage between the
26514 * inode mounted on and the root directory of the mounted file system.
26515 */
26516 while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
26517     /* The inode is indeed mounted on. */
26518     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
26519         if (sp->s_imount == rip) {
26520             /* Release the inode mounted on. Replace by the
26521             * inode of the root inode of the mounted device.
26522             */
26523             put_inode(rip);
26524             rip = get_inode(sp->s_dev, ROOT_INODE);
26525             break;
26526         }
26527     }
26528 }
26529 return(rip);      /* return pointer to inode's component */
26530 }

26531 /*=====
26532 *                         search_dir
26533 *=====
26534 */
26535 PUBLIC int search_dir(lptr, string, numb, flag)

```

```

26536 register struct inode *ldir_ptr; /* ptr to inode for dir to search */
26537 char string[NAME_MAX];           /* component to search for */
26538 ino_t *numb;                  /* pointer to inode number */
26539 int flag;                    /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
26540 {
26541 /* This function searches the directory whose inode is pointed to by 'ldir':
26542 * if (flag == ENTER) enter 'string' in the directory with inode # '*numb';
26543 * if (flag == DELETE) delete 'string' from the directory;
26544 * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
26545 * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
26546 *
26547 *   if 'string' is dot1 or dot2, no access permissions are checked.
26548 */
26549
26550 register struct direct *dp = NULL;
26551 register struct buf *bp = NULL;
26552 int i, r, e_hit, t, match;
26553 mode_t bits;
26554 off_t pos;
26555 unsigned new_slots, old_slots;
26556 block_t b;
26557 struct super_block *sp;
26558 int extended = 0;
26559
26560 /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
26561 if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);
26562
26563 r = OK;
26564
26565 if (flag != IS_EMPTY) {
26566     bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);
26567
26568     if (string == dot1 || string == dot2) {
26569         if (flag != LOOK_UP) r = read_only(ldir_ptr);
26570             /* only a writable device is required. */
26571     }
26572     else r = forbidden(ldir_ptr, bits); /* check access permissions */
26573 }
26574 if (r != OK) return(r);
26575
26576 /* Step through the directory one block at a time. */
26577 old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
26578 new_slots = 0;
26579 e_hit = FALSE;
26580 match = 0;                      /* set when a string match occurs */
26581
26582 for (pos = 0; pos < ldir_ptr->i_size; pos += ldir_ptr->i_sp->s_block_size) {
26583     b = read_map(ldir_ptr, pos); /* get block number */
26584
26585     /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
26586     bp = get_block(ldir_ptr->i_dev, b, NORMAL); /* get a dir block */
26587
26588     if (bp == NO_BLOCK)
26589         panic(FILE, "get_block returned NO_BLOCK", NO_NUM);
26590
26591     /* Search a directory block. */
26592     for (dp = &bp->b_dir[0];
26593         dp < &bp->b_dir[NR_DIR_ENTRIES(ldir_ptr->i_sp->s_block_size)];
26594         dp++) {
26595         if (++new_slots > old_slots) { /* not found, but room left */
26596             if (flag == ENTER) e_hit = TRUE;
26597             break;
26598         }
26599
26600         /* Match occurs if string found. */
26601         if (flag != ENTER && dp->d_ino != 0) {
26602             if (flag == IS_EMPTY) {

```

```

26603             /* If this test succeeds, dir is not empty. */
26604             if (strcmp(dp->d_name, ".") != 0 &&
26605                 strcmp(dp->d_name, "..") != 0) match = 1;
26606         } else {
26607             if (strncmp(dp->d_name, string, NAME_MAX) == 0) {
26608                 match = 1;
26609             }
26610         }
26611     }
26612
26613     if (match) {
26614         /* LOOK_UP or DELETE found what it wanted. */
26615         r = OK;
26616         if (flag == IS_EMPTY) r = ENOTEMPTY;
26617         else if (flag == DELETE) {
26618             /* Save d_ino for recovery. */
26619             t = NAME_MAX - sizeof(ino_t);
26620             *((ino_t *) &dp->d_name[t]) = dp->d_ino;
26621             dp->d_ino = 0; /* erase entry */
26622             bp->b_dirt = DIRTY;
26623             ldir_ptr->i_update |= CTIME | MTIME;
26624             ldir_ptr->i_dirt = DIRTY;
26625         } else {
26626             sp = ldir_ptr->i_sp; /* 'flag' is LOOK_UP */
26627             *numb = conv4(sp->s_native, (int) dp->d_ino);
26628         }
26629         put_block(bp, DIRECTORY_BLOCK);
26630         return(r);
26631     }
26632
26633     /* Check for free slot for the benefit of ENTER. */
26634     if (flag == ENTER && dp->d_ino == 0) {
26635         e_hit = TRUE; /* we found a free slot */
26636         break;
26637     }
26638 }
26639
26640 /* The whole block has been searched or ENTER has a free slot. */
26641 if (e_hit) break; /* e_hit set if ENTER can be performed now */
26642 put_block(bp, DIRECTORY_BLOCK); /* otherwise, continue searching dir */
26643 }
26644
26645 /* The whole directory has now been searched. */
26646 if (flag != ENTER) {
26647     return(flag == IS_EMPTY ? OK : ENOENT);
26648 }
26649
26650 /* This call is for ENTER. If no free slot has been found so far, try to
26651   * extend directory.
26652   */
26653 if (e_hit == FALSE) { /* directory is full and no room left in last block */
26654     new_slots++; /* increase directory size by 1 entry */
26655     if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
26656     if ((bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
26657         return(err_code);
26658     dp = &bp->b_dir[0];
26659     extended = 1;
26660 }
26661
26662 /* 'bp' now points to a directory block with space. 'dp' points to slot. */
26663 (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
26664 for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
26665 sp = ldir_ptr->i_sp;
26666 dp->d_ino = conv4(sp->s_native, (int) *numb);
26667 bp->b_dirt = DIRTY;
26668 put_block(bp, DIRECTORY_BLOCK);
26669 ldir_ptr->i_update |= CTIME | MTIME; /* mark mtime for update later */
26670 ldir_ptr->i_dirt = DIRTY;

```

```

26671     if (new_slots > old_slots) {
26672         ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
26673         /* Send the change to disk if the directory is extended. */
26674         if (extended) rw_inode(ldir_ptr, WRITING);
26675     }
26676     return(OK);
26677 }

+++++
servers/fs/mount.c
+++++

26700 /* This file performs the MOUNT and UMOUNT system calls.
26701 *
26702 * The entry points into this file are
26703 *   do_mount: perform the MOUNT system call
26704 *   do_umount: perform the UMOUNT system call
26705 */
26706
26707 #include "fs.h"
26708 #include <fcntl.h>
26709 #include <minix/com.h>
26710 #include <sys/stat.h>
26711 #include "buf.h"
26712 #include "file.h"
26713 #include "fproc.h"
26714 #include "inode.h"
26715 #include "param.h"
26716 #include "super.h"
26717
26718 FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path) );
26719
26720 /*=====
26721 *          do_mount
26722 *=====
26723 PUBLIC int do_mount()
26724 {
26725 /* Perform the mount(name, mfile, rd_only) system call. */
26726
26727     register struct inode *rip, *root_ip;
26728     struct super_block *xp, *sp;
26729     dev_t dev;
26730     mode_t bits;
26731     int rdir, mdir;           /* TRUE iff {root|mount} file is dir */
26732     int r, found;
26733
26734 /* Only the super-user may do MOUNT. */
26735     if (!super_user) return(EPERM);
26736
26737 /* If 'name' is not for a block special file, return error. */
26738     if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
26739     if ((dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
26740
26741 /* Scan super block table to see if dev already mounted & find a free slot.*/
26742     sp = NIL_SUPER;
26743     found = FALSE;
26744     for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
26745         if (xp->s_dev == dev) found = TRUE;      /* is it mounted already? */
26746         if (xp->s_dev == NO_DEV) sp = xp;        /* record free slot */
26747     }
26748     if (found) return(EBUSY);      /* already mounted */
26749     if (sp == NIL_SUPER) return(ENFILE); /* no super block available */
26750
26751 /* Open the device the file system lives on. */
26752     if (dev_open(dev, who, m_in.rd_only ? R_BIT : (R_BIT|W_BIT)) != OK)
26753         return(EINVAL);
26754

```

```

26755 /* Make the cache forget about blocks it has open on the filesystem */
26756 (void) do_sync();
26757 invalidate(dev);
26758
26759 /* Fill in the super block. */
26760 sp->s_dev = dev; /* read_super() needs to know which dev */
26761 r = read_super(sp);
26762
26763 /* Is it recognized as a Minix filesystem? */
26764 if (r != OK) {
26765     dev_close(dev);
26766     sp->s_dev = NO_DEV;
26767     return(r);
26768 }
26769
26770 /* Now get the inode of the file to be mounted on. */
26771 if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
26772     dev_close(dev);
26773     sp->s_dev = NO_DEV;
26774     return(err_code);
26775 }
26776 if ((rip = eat_path(user_path)) == NIL_INODE) {
26777     dev_close(dev);
26778     sp->s_dev = NO_DEV;
26779     return(err_code);
26780 }
26781
26782 /* It may not be busy. */
26783 r = OK;
26784 if (rip->i_count > 1) r = EBUSY;
26785
26786 /* It may not be special. */
26787 bits = rip->i_mode & I_TYPE;
26788 if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;
26789
26790 /* Get the root inode of the mounted file system. */
26791 root_ip = NIL_INODE; /* if 'r' not OK, make sure this is defined */
26792 if (r == OK) {
26793     if ((root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
26794 }
26795 if (root_ip != NIL_INODE && root_ip->i_mode == 0) {
26796     r = EINVAL;
26797 }
26798
26799 /* File types of 'rip' and 'root_ip' may not conflict. */
26800 if (r == OK) {
26801     mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY); /* TRUE iff dir */
26802     rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
26803     if (!mdir && rdir) r = EISDIR;
26804 }
26805
26806 /* If error, return the super block and both inodes; release the maps. */
26807 if (r != OK) {
26808     put_inode(rip);
26809     put_inode(root_ip);
26810     (void) do_sync();
26811     invalidate(dev);
26812     dev_close(dev);
26813     sp->s_dev = NO_DEV;
26814     return(r);
26815 }
26816
26817 /* Nothing else can go wrong. Perform the mount. */
26818 rip->i_mount = I_MOUNT; /* this bit says the inode is mounted on */
26819 sp->s_imount = rip;
26820 sp->s_isup = root_ip;
26821 sp->s_rd_only = m_in.rd_only;
26822 return(OK);

```

```

26823 }
26825 /*=====
26826 *          do_umount
26827 *=====
26828 PUBLIC int do_umount()
26829 {
26830 /* Perform the umount(name) system call. */
26831     dev_t dev;
26832
26833 /* Only the super-user may do UMOUNT. */
26834     if (!super_user) return(EPERM);
26835
26836 /* If 'name' is not for a block special file, return error. */
26837     if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
26838     if ((dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
26839
26840     return(unmount(dev));
26841 }

26843 /*=====
26844 *          unmount
26845 *=====
26846 PUBLIC int unmount(dev)
26847 Dev_t dev;
26848 {
26849 /* Unmount a file system by device number. */
26850     register struct inode *rip;
26851     struct super_block *sp, *spl;
26852     int count;
26853
26854 /* See if the mounted device is busy. Only 1 inode using it should be
26855 * open -- the root inode -- and that inode only 1 time.
26856 */
26857     count = 0;
26858     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
26859         if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
26860     if (count > 1) return(EBUSY); /* can't umount a busy file system */
26861
26862 /* Find the super block. */
26863     sp = NIL_SUPER;
26864     for (spl = &super_block[0]; spl < &super_block[NR_SUPERS]; spl++) {
26865         if (spl->s_dev == dev) {
26866             sp = spl;
26867             break;
26868         }
26869     }
26870
26871 /* Sync the disk, and invalidate cache. */
26872     (void) do_sync();           /* force any cached blocks out of memory */
26873     invalidate(dev);          /* invalidate cache entries for this dev */
26874     if (sp == NIL_SUPER) {
26875         return(EINVAL);
26876     }
26877
26878 /* Close the device the file system lives on. */
26879     dev_close(dev);
26880
26881 /* Finish off the unmount. */
26882     sp->s_imount->i_mount = NO_MOUNT; /* inode returns to normal */
26883     put_inode(sp->s_imount);        /* release the inode mounted on */
26884     put_inode(sp->s_isup);         /* release the root inode of the mounted fs */
26885     sp->s_imount = NIL_INODE;
26886     sp->s_dev = NO_DEV;
26887     return(OK);
26888 }

```

```

26890 /*-----*
26891 *----- name_to_dev -----*
26892 *-----*/
26893 PRIVATE dev_t name_to_dev(path)
26894 char *path; /* pointer to path name */
26895 {
26896 /* Convert the block special file 'path' to a device number. If 'path'
26897 * is not a block special file, return error code in 'err_code'.
26898 */
26899
26900 register struct inode *rip;
26901 register dev_t dev;
26902
26903 /* If 'path' can't be opened, give up immediately. */
26904 if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);
26905
26906 /* If 'path' is not a block special file, return error. */
26907 if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
26908     err_code = ENOTBLK;
26909     put_inode(rip);
26910     return(NO_DEV);
26911 }
26912
26913 /* Extract the device number. */
26914 dev = (dev_t) rip->i_zone[0];
26915 put_inode(rip);
26916 return(dev);
26917 }
```

```
+++++
servers/fs/link.c
+++++
```

```

27000 /* This file handles the LINK and UNLINK system calls. It also deals with
27001 * deallocating the storage used by a file when the last UNLINK is done to a
27002 * file and the blocks must be returned to the free block pool.
27003 *
27004 * The entry points into this file are
27005 * do_link: perform the LINK system call
27006 * do_unlink: perform the UNLINK and RMDIR system calls
27007 * do_rename: perform the RENAME system call
27008 * truncate: release all the blocks associated with an inode
27009 */
27010
27011 #include "fs.h"
27012 #include <sys/stat.h>
27013 #include <string.h>
27014 #include <minix/com.h>
27015 #include <minix/callnr.h>
27016 #include "buf.h"
27017 #include "file.h"
27018 #include "fproc.h"
27019 #include "inode.h"
27020 #include "param.h"
27021 #include "super.h"
27022
27023 #define SAME 1000
27024 FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
27025                         char dir_name[NAME_MAX]) );
27026
27027 FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirlp, struct inode *rip,
27028                         char file_name[NAME_MAX]) );
27029
27030
```

```

27031 /*-----*
27032 * do_link
27033 *-----*/
27034 PUBLIC int do_link()
27035 {
27036 /* Perform the link(name1, name2) system call. */
27037
27038     register struct inode *ip, *rip;
27039     register int r;
27040     char string[NAME_MAX];
27041     struct inode *new_ip;
27042
27043 /* See if 'name' (file to be linked) exists. */
27044     if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
27045     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
27046
27047 /* Check to see if the file has maximum number of links already. */
27048     r = OK;
27049     if (rip->i_nlinks >= (rip->i_sp->s_version == V1 ? CHAR_MAX : SHRT_MAX))
27050         r = EMLINK;
27051
27052 /* Only super_user may link to directories. */
27053     if (r == OK)
27054         if ((rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
27055
27056 /* If error with 'name', return the inode. */
27057     if (r != OK) {
27058         put_inode(rip);
27059         return(r);
27060     }
27061
27062 /* Does the final directory of 'name2' exist? */
27063     if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
27064         put_inode(rip);
27065         return(err_code);
27066     }
27067     if ((ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;
27068
27069 /* If 'name2' exists in full (even if no space) set 'r' to error. */
27070     if (r == OK) {
27071         if ((new_ip = advance(ip, string)) == NIL_INODE) {
27072             r = err_code;
27073             if (r == ENOENT) r = OK;
27074         } else {
27075             put_inode(new_ip);
27076             r = EEXIST;
27077         }
27078     }
27079
27080 /* Check for links across devices. */
27081     if (r == OK)
27082         if (rip->i_dev != ip->i_dev) r = EXDEV;
27083
27084 /* Try to link. */
27085     if (r == OK)
27086         r = search_dir(ip, string, &rip->i_num, ENTER);
27087
27088 /* If success, register the linking. */
27089     if (r == OK) {
27090         rip->i_nlinks++;
27091         rip->i_update |= CTIME;
27092         rip->i_dirt = DIRTY;
27093     }
27094
27095 /* Done. Release both inodes. */
27096     put_inode(rip);
27097     put_inode(ip);

```

```

27098     return(r);
27099 }

27101 /*=====
27102 *          do_unlink
27103 *=====
27104 PUBLIC int do_unlink()
27105 {
27106 /* Perform the unlink(name) or rmdir(name) system call. The code for these two
27107 * is almost the same. They differ only in some condition testing. Unlink()
27108 * may be used by the superuser to do dangerous things; rmdir() may not.
27109 */
27110
27111     register struct inode *rip;
27112     struct inode *rldirp;
27113     int r;
27114     char string[NAME_MAX];
27115
27116 /* Get the last directory in the path. */
27117     if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
27118     if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
27119         return(err_code);
27120
27121 /* The last directory exists. Does the file also exist? */
27122     r = OK;
27123     if ( (rip = advance(rldirp, string)) == NIL_INODE) r = err_code;
27124
27125 /* If error, return inode. */
27126     if (r != OK) {
27127         put_inode(rldirp);
27128         return(r);
27129     }
27130
27131 /* Do not remove a mount point. */
27132     if (rip->i_num == ROOT_INODE) {
27133         put_inode(rldirp);
27134         put_inode(rip);
27135         return(EBUSY);
27136     }
27137
27138 /* Now test if the call is allowed, separately for unlink() and rmdir(). */
27139     if (call_nr == UNLINK) {
27140         /* Only the su may unlink directories, but the su can unlink any dir.*/
27141         if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
27142
27143         /* Don't unlink a file if it is the root of a mounted file system. */
27144         if (rip->i_num == ROOT_INODE) r = EBUSY;
27145
27146         /* Actually try to unlink the file; fails if parent is mode 0 etc. */
27147         if (r == OK) r = unlink_file(rldirp, rip, string);
27148
27149     } else {
27150         r = remove_dir(rldirp, rip, string); /* call is RMDIR */
27151     }
27152
27153 /* If unlink was possible, it has been done, otherwise it has not. */
27154     put_inode(rip);
27155     put_inode(rldirp);
27156     return(r);
27157 }

27159 /*=====
27160 *          do_rename
27161 *=====
27162 PUBLIC int do_rename()
27163 {
27164 /* Perform the rename(name1, name2) system call. */
27165

```

```

27166 struct inode *old_dirp, *old_ip;      /* ptrs to old dir, file inodes */
27167 struct inode *new_dirp, *new_ip;      /* ptrs to new dir, file inodes */
27168 struct inode *new_superdirp, *next_new_superdirp;
27169 int r = OK;                         /* error flag; initially no error */
27170 int odir, ndir;                     /* TRUE iff {old|new} file is dir */
27171 int same_pdir;                     /* TRUE iff parent dirs are the same */
27172 char old_name[NAME_MAX], new_name[NAME_MAX];
27173 ino_t numb;
27174 int r1;
27175
27176 /* See if 'name1' (existing file) exists. Get dir and file inodes. */
27177 if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
27178 if ((old_dirp = last_dir(user_path, old_name)) == NIL_INODE) return(err_code);
27179
27180 if ((old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;
27181
27182 /* See if 'name2' (new name) exists. Get dir and file inodes. */
27183 if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) r = err_code;
27184 if ((new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
27185 new_ip = advance(new_dirp, new_name); /* not required to exist */
27186
27187 if (old_ip != NIL_INODE)
27188     odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY); /* TRUE iff dir */
27189
27190 /* If it is ok, check for a variety of possible errors. */
27191 if (r == OK) {
27192     same_pdir = (old_dirp == new_dirp);
27193
27194     /* The old inode must not be a superdirectory of the new last dir. */
27195     if (odir && !same_pdir) {
27196         dup_inode(new_superdirp = new_dirp);
27197         while (TRUE) { /* may hang in a file system loop */
27198             if (new_superdirp == old_ip) {
27199                 r = EINVAL;
27200                 break;
27201             }
27202             next_new_superdirp = advance(new_superdirp, dot2);
27203             put_inode(new_superdirp);
27204             if (next_new_superdirp == new_superdirp)
27205                 break; /* back at system root directory */
27206             new_superdirp = next_new_superdirp;
27207             if (new_superdirp == NIL_INODE) {
27208                 /* Missing ".." entry. Assume the worst. */
27209                 r = EINVAL;
27210                 break;
27211             }
27212         }
27213         put_inode(new_superdirp);
27214     }
27215
27216     /* The old or new name must not be . or .. */
27217     if (strcmp(old_name, ".") == 0 || strcmp(old_name, "..") == 0 ||
27218         strcmp(new_name, ".") == 0 || strcmp(new_name, "..") == 0) r = EINVAL;
27219
27220     /* Both parent directories must be on the same device. */
27221     if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;
27222
27223     /* Parent dirs must be writable, searchable and on a writable device */
27224     if ((r1 = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
27225         (r1 = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = r1;
27226
27227     /* Some tests apply only if the new path exists. */
27228     if (new_ip == NIL_INODE) {
27229         /* don't rename a file with a file system mounted on it. */
27230         if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
27231         if (odir && new_dirp->i_nlinks >=
27232             (new_dirp->i_sp->s_version == V1 ? CHAR_MAX : SHRT_MAX) &&

```

```

27233             !same_pdir && r == OK) r = EMLINK;
27234     } else {
27235         if (old_ip == new_ip) r = SAME; /* old=new */
27236
27237         /* has the old file or new file a file system mounted on it? */
27238         if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;
27239
27240         ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
27241         if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
27242         if (odir == FALSE && ndir == TRUE) r = EISDIR;
27243     }
27244 }
27245
27246 /* If a process has another root directory than the system root, we might
27247 * "accidentally" be moving it's working directory to a place where it's
27248 * root directory isn't a super directory of it anymore. This can make
27249 * the function chroot useless. If chroot will be used often we should
27250 * probably check for it here.
27251 */
27252
27253 /* The rename will probably work. Only two things can go wrong now:
27254 * 1. being unable to remove the new file. (when new file already exists)
27255 * 2. being unable to make the new directory entry. (new file doesn't exists)
27256 * [directory has to grow by one block and cannot because the disk
27257 * is completely full].
27258 */
27259 if (r == OK) {
27260     if (new_ip != NIL_INODE) {
27261         /* There is already an entry for 'new'. Try to remove it. */
27262         if (odir)
27263             r = remove_dir(new_dirp, new_ip, new_name);
27264         else
27265             r = unlink_file(new_dirp, new_ip, new_name);
27266     }
27267     /* if r is OK, the rename will succeed, while there is now an
27268     * unused entry in the new parent directory.
27269     */
27270 }
27271
27272 if (r == OK) {
27273     /* If the new name will be in the same parent directory as the old one,
27274     * first remove the old name to free an entry for the new name,
27275     * otherwise first try to create the new name entry to make sure
27276     * the rename will succeed.
27277     */
27278     numb = old_ip->i_num;           /* inode number of old file */
27279
27280     if (same_pdir) {
27281         r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
27282             /* shouldn't go wrong. */
27283         if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
27284     } else {
27285         r = search_dir(new_dirp, new_name, &numb, ENTER);
27286         if (r == OK)
27287             (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
27288     }
27289
27290     /* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked
27291     * for update in search_dir.
27292     */
27293
27294     if (r == OK && odir && !same_pdir) {
27295         /* Update the .. entry in the directory (still points to old_dirp). */
27296         numb = new_dirp->i_num;
27297         (void) unlink_file(old_ip, NIL_INODE, dot2);
27298         if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
27299             /* New link created. */

```

```

27300             new_dirp->i_nlinks++;
27301             new_dirp->i_dirt = DIRTY;
27302         }
27303     }
27304
27305     /* Release the inodes. */
27306     put_inode(old_dirp);
27307     put_inode(old_ip);
27308     put_inode(new_dirp);
27309     put_inode(new_ip);
27310     return(r == SAME ? OK : r);
27311 }

27313 /*-----*
27314 *          truncate
27315 *-----*/
27316 PUBLIC void truncate(rip)
27317 register struct inode *rip;      /* pointer to inode to be truncated */
27318 {
27319 /* Remove all the zones from the inode 'rip' and mark it dirty. */
27320
27321 register block_t b;
27322 zone_t z, zone_size, z1;
27323 off_t position;
27324 int i, scale, file_type, waspipe, single, nr_indirects;
27325 struct buf *bp;
27326 dev_t dev;
27327
27328 file_type = rip->i_mode & I_TYPE;      /* check to see if file is special */
27329 if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
27330 dev = rip->i_dev;                  /* device on which inode resides */
27331 scale = rip->i_sp->s_log_zone_size;
27332 zone_size = (zone_t) rip->i_sp->s_block_size << scale;
27333 nr_indirects = rip->i_nindirs;
27334
27335 /* Pipes can shrink, so adjust size to make sure all zones are removed. */
27336 waspipe = rip->i_pipe == I_PIPE;      /* TRUE is this was a pipe */
27337 if (waspipe) rip->i_size = PIPE_SIZE(rip->i_sp->s_block_size);
27338
27339 /* Step through the file a zone at a time, finding and freeing the zones. */
27340 for (position = 0; position < rip->i_size; position += zone_size) {
27341     if ((b = read_map(rip, position)) != NO_BLOCK) {
27342         z = (zone_t) b >> scale;
27343         free_zone(dev, z);
27344     }
27345 }
27346
27347 /* All the data zones have been freed. Now free the indirect zones. */
27348 rip->i_dirt = DIRTY;
27349 if (waspipe) {
27350     wipe_inode(rip);           /* clear out inode for pipes */
27351     return;                   /* indirect slots contain file positions */
27352 }
27353 single = rip->i_ndzones;
27354 free_zone(dev, rip->i_zone[single]); /* single indirect zone */
27355 if ((z = rip->i_zone[single+1]) != NO_ZONE) {
27356     /* Free all the single indirect zones pointed to by the double. */
27357     b = (block_t) z << scale;
27358     bp = get_block(dev, b, NORMAL); /* get double indirect zone */
27359     for (i = 0; i < nr_indirects; i++) {
27360         z1 = rd_indir(bp, i);
27361         free_zone(dev, z1);
27362     }
27363
27364     /* Now free the double indirect zone itself. */
27365     put_block(bp, INDIRECT_BLOCK);
27366     free_zone(dev, z);

```

```

27367 }
27368
27369 /* Leave zone numbers for de(1) to recover file after an unlink(2). */
27370 }

27372 /*=====
27373 *          remove_dir
27374 *=====
27375 PRIVATE int remove_dir(rldirp, rip, dir_name)
27376 struct inode *rldirp;           /* parent directory */
27377 struct inode *rip;            /* directory to be removed */
27378 char dir_name[NAME_MAX];      /* name of directory to be removed */
27379 {
27380     /* A directory file has to be removed. Five conditions have to met:
27381     *   - The file must be a directory
27382     *   - The directory must be empty (except for . and ..)
27383     *   - The final component of the path must not be . or ..
27384     *   - The directory must not be the root of a mounted file system
27385     *   - The directory must not be anybody's root/working directory
27386     */
27387
27388     int r;
27389     register struct fproc *rfp;
27390
27391     /* search_dir checks that rip is a directory too. */
27392     if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;
27393
27394     if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0) return(EINVAL);
27395     if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */
27396
27397     for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
27398         if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
27399             /* can't remove anybody's working dir */
27400
27401     /* Actually try to unlink the file; fails if parent is mode 0 etc. */
27402     if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;
27403
27404     /* Unlink . and .. from the dir. The super user can link and unlink any dir,
27405     * so don't make too many assumptions about them.
27406     */
27407     (void) unlink_file(rip, NIL_INODE, dot1);
27408     (void) unlink_file(rip, NIL_INODE, dot2);
27409     return(OK);
27410 }

27412 /*=====
27413 *          unlink_file
27414 *=====
27415 PRIVATE int unlink_file(dirp, rip, file_name)
27416 struct inode *dirp;           /* parent directory of file */
27417 struct inode *rip;            /* inode of file, may be NIL_INODE too. */
27418 char file_name[NAME_MAX];      /* name of file to be removed */
27419 {
27420     /* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */
27421
27422     ino_t numb;                /* inode number */
27423     int r;
27424
27425     /* If rip is not NIL_INODE, it is used to get faster access to the inode. */
27426     if (rip == NIL_INODE) {
27427         /* Search for file in directory and try to get its inode. */
27428         err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
27429         if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
27430         if (err_code != OK || rip == NIL_INODE) return(err_code);
27431     } else {
27432         dup_inode(rip);           /* inode will be returned with put_inode */
27433     }

```

```

27434
27435     r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);
27436
27437     if (r == OK) {
27438         rip->i_nlinks--;           /* entry deleted from parent's dir */
27439         rip->i_update |= CTIME;
27440         rip->i_dirt = DIRTY;
27441     }
27442
27443     put_inode(rip);
27444     return(r);
27445 }

+++++
servers/fs/stadir.c
+++++

27500 /* This file contains the code for performing four system calls relating to
27501 * status and directories.
27502 *
27503 * The entry points into this file are
27504 *   do_chdir: perform the CHDIR system call
27505 *   do_chroot: perform the CHROOT system call
27506 *   do_stat: perform the STAT system call
27507 *   do_fstat: perform the FSTAT system call
27508 *   do_fstatfs: perform the FSTATFS system call
27509 */
27510
27511 #include "fs.h"
27512 #include <sys/stat.h>
27513 #include <sys/statfs.h>
27514 #include <minix/com.h>
27515 #include "file.h"
27516 #include "fproc.h"
27517 #include "inode.h"
27518 #include "param.h"
27519 #include "super.h"
27520
27521 FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr, int len));
27522 FORWARD _PROTOTYPE( int change_into, (struct inode **iip, struct inode *ip));
27523 FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp *filp_ptr,
27524                               char *user_addr));
27525
27526 /*=====
27527 *          do_fchdir
27528 *=====
27529 PUBLIC int do_fchdir()
27530 {
27531     /* Change directory on already-opened fd. */
27532     struct filp *rfilp;
27533
27534     /* Is the file descriptor valid? */
27535     if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
27536     return change_into(&fp->fp_workdir, rfilp->filp_ino);
27537 }

27538 /*=====
27539 *          do_chdir
27540 *=====
27541 PUBLIC int do_chdir()
27542 {
27543     /* Change directory. This function is also called by MM to simulate a chdir
27544     * in order to do EXEC, etc. It also changes the root directory, the uids and

```

```

27546 * gids, and the umask.
27547 */
27548
27549     int r;
27550     register struct fproc *rfp;
27551
27552     if (who == PM_PROC_NR) {
27553         rfp = &fproc[m_in.slot1];
27554         put_inode(fp->fp_rootdir);
27555         dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
27556         put_inode(fp->fp_workdir);
27557         dup_inode(fp->fp_workdir = rfp->fp_workdir);
27558
27559         /* MM uses access() to check permissions. To make this work, pretend
27560         * that the user's real ids are the same as the user's effective ids.
27561         * FS calls other than access() do not use the real ids, so are not
27562         * affected.
27563         */
27564         fp->fp_realuid =
27565         fp->fp_effuid = rfp->fp_effuid;
27566         fp->fp_realgid =
27567         fp->fp_effgid = rfp->fp_effgid;
27568         fp->fp_umask = rfp->fp_umask;
27569
27570     }
27571
27572     /* Perform the chdir(name) system call. */
27573     r = change(&fp->fp_workdir, m_in.name, m_in.name_length);
27574     return(r);
27575 }

27576 /*=====
27577 *          do_chroot
27578 *=====
27579 PUBLIC int do_chroot()
27580 {
27581     /* Perform the chroot(name) system call. */
27582
27583     register int r;
27584
27585     if (!super_user) return(EPERM);      /* only su may chroot() */
27586     r = change(&fp->fp_rootdir, m_in.name, m_in.name_length);
27587     return(r);
27588 }

27589 /*=====
27590 *          change
27591 *=====
27592 PRIVATE int change(iip, name_ptr, len)
27593 struct inode **iip;           /* pointer to the inode pointer for the dir */
27594 char *name_ptr;             /* pointer to the directory name to change to */
27595 int len;                   /* length of the directory name string */
27596 {
27597     /* Do the actual work for chdir() and chroot(). */
27598     struct inode *rip;
27599
27600     /* Try to open the new directory. */
27601     if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
27602     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
27603     return change_into(iip, rip);
27604 }

27605 /*=====
27606 *          change_int
27607 *=====
27608 PRIVATE int change_int(iip, rip)
27609 struct inode **iip;           /* pointer to the inode pointer for the dir */
27610
27611
27612

```

```

27613 struct inode *rip;           /* this is what the inode has to become */
27614 {
27615     register int r;
27616
27617     /* It must be a directory and also be searchable. */
27618     if ((rip->i_mode & I_TYPE) != I_DIRECTORY)
27619         r = ENOTDIR;
27620     else
27621         r = forbidden(rip, X_BIT);      /* check if dir is searchable */
27622
27623     /* If error, return inode. */
27624     if (r != OK) {
27625         put_inode(rip);
27626         return(r);
27627     }
27628
27629     /* Everything is OK. Make the change. */
27630     put_inode(*iip);             /* release the old directory */
27631     *iip = rip;                /* acquire the new one */
27632     return(OK);
27633 }

27635 /*=====
27636 *          do_stat
27637 *=====
27638 PUBLIC int do_stat()
27639 {
27640     /* Perform the stat(name, buf) system call. */
27641
27642     register struct inode *rip;
27643     register int r;
27644
27645     /* Both stat() and fstat() use the same routine to do the real work. That
27646     * routine expects an inode, so acquire it temporarily.
27647     */
27648     if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
27649     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
27650     r = stat_inode(rip, NIL_FILP, m_in.name2);    /* actually do the work.*/
27651     put_inode(rip);                            /* release the inode */
27652     return(r);
27653 }

27655 /*=====
27656 *          do_fstat
27657 *=====
27658 PUBLIC int do_fstat()
27659 {
27660     /* Perform the fstat(fd, buf) system call. */
27661
27662     register struct filp *rfilep;
27663
27664     /* Is the file descriptor valid? */
27665     if ((rfilep = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
27666
27667     return(stat_inode(rfilep->filp_ino, rfilep, m_in.buffer));
27668 }

27670 /*=====
27671 *          stat_inode
27672 *=====
27673 PRIVATE int stat_inode(rip, fil_ptr, user_addr)
27674     register struct inode *rip;      /* pointer to inode to stat */
27675     struct filp *fil_ptr;          /* filp pointer, supplied by 'fstat' */
27676     char *user_addr;              /* user space address where stat buf goes */
27677
27678     /* Common code for stat and fstat system calls. */
27679

```

```

27680     struct stat statbuf;
27681     mode_t mo;
27682     int r, s;
27683
27684     /* Update the atime, ctime, and mtime fields in the inode, if need be. */
27685     if (rip->i_update) update_times(rip);
27686
27687     /* Fill in the statbuf struct. */
27688     mo = rip->i_mode & I_TYPE;
27689
27690     /* true iff special */
27691     s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);
27692
27693     statbuf.st_dev = rip->i_dev;
27694     statbuf.st_ino = rip->i_num;
27695     statbuf.st_mode = rip->i_mode;
27696     statbuf.st_nlink = rip->i_nlinks;
27697     statbuf.st_uid = rip->i_uid;
27698     statbuf.st_gid = rip->i_gid;
27699     statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
27700     statbuf.st_size = rip->i_size;
27701
27702     if (rip->i_pipe == I_PIPE) {
27703         statbuf.st_mode &= ~I_REGULAR; /* wipe out I_REGULAR bit for pipes */
27704         if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
27705             statbuf.st_size -= fil_ptr->filp_pos;
27706     }
27707
27708     statbuf.st_atime = rip->i_atime;
27709     statbuf.st_mtime = rip->i_mtime;
27710     statbuf.st_ctime = rip->i_ctime;
27711
27712     /* Copy the struct to user space. */
27713     r = sys_datacopy(FS_PROC_NR, (vir_bytes) &statbuf,
27714                         who, (vir_bytes) user_addr, (phys_bytes) sizeof(statbuf));
27715
27716 }

27718 /*=====
27719 *          do_fstatfs
27720 *=====
27721 PUBLIC int do_fstatfs()
27722 {
27723     /* Perform the fstatfs(fd, buf) system call. */
27724     struct statfs st;
27725     register struct filp *rfilp;
27726     int r;
27727
27728     /* Is the file descriptor valid? */
27729     if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
27730
27731     st.f_bsize = rfilp->filp_ino->i_sp->s_block_size;
27732
27733     r = sys_datacopy(FS_PROC_NR, (vir_bytes) &st,
27734                         who, (vir_bytes) m_in.buffer, (phys_bytes) sizeof(st));
27735
27736     return(r);
27737 }

*****+
*****+ servers/fs/protect.c
*****+
27800 /* This file deals with protection in the file system. It contains the code
27801 * for four system calls that relate to protection.

```

```

27802 /*
27803 * The entry points into this file are
27804 * do_chmod: perform the CHMOD system call
27805 * do_chown: perform the CHOWN system call
27806 * do_umask: perform the UMASK system call
27807 * do_access: perform the ACCESS system call
27808 * forbidden: check to see if a given access is allowed on a given inode
27809 */
27810
27811 #include "fs.h"
27812 #include <unistd.h>
27813 #include <minix/callnr.h>
27814 #include "buf.h"
27815 #include "file.h"
27816 #include "fproc.h"
27817 #include "inode.h"
27818 #include "param.h"
27819 #include "super.h"
27820
27821 /*=====
27822 *                                do_chmod
27823 *=====
27824 PUBLIC int do_chmod()
27825 {
27826 /* Perform the chmod(name, mode) system call. */
27827
27828     register struct inode *rip;
27829     register int r;
27830
27831 /* Temporarily open the file. */
27832     if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
27833     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
27834
27835 /* Only the owner or the super_user may change the mode of a file.
27836 * No one may change the mode of a file on a read-only file system.
27837 */
27838     if (rip->i_uid != fp->fp_effuid && !super_user)
27839         r = EPERM;
27840     else
27841         r = read_only(rip);
27842
27843 /* If error, return inode. */
27844     if (r != OK) {
27845         put_inode(rip);
27846         return(r);
27847     }
27848
27849 /* Now make the change. Clear setgid bit if file is not in caller's grp */
27850     rip->i_mode = (rip->i_mode & ~ALL_MODES) | (m_in.mode & ALL_MODES);
27851     if (!super_user && rip->i_gid != fp->fp_effgid) rip->i_mode &= ~I_SET_GID_BIT;
27852     rip->i_update |= CTIME;
27853     rip->i_dirt = DIRTY;
27854
27855     put_inode(rip);
27856     return(OK);
27857 }
27858
27859 /*=====
27860 *                                do_chown
27861 *=====
27862 PUBLIC int do_chown()
27863 {
27864 /* Perform the chown(name, owner, group) system call. */
27865
27866     register struct inode *rip;
27867     register int r;
27868

```

```

27869 /* Temporarily open the file. */
27870 if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
27871 if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
27872
27873 /* Not permitted to change the owner of a file on a read-only file sys. */
27874 r = read_only(rip);
27875 if (r == OK) {
27876     /* FS is R/W. Whether call is allowed depends on ownership, etc. */
27877     if (super_user) {
27878         /* The super user can do anything. */
27879         rip->i_uid = m_in.owner;           /* others later */
27880     } else {
27881         /* Regular users can only change groups of their own files. */
27882         if (rip->i_uid != fp->fp_effuid) r = EPERM;
27883         if (rip->i_uid != m_in.owner) r = EPERM; /* no giving away */
27884         if (fp->fp_effgid != m_in.group) r = EPERM;
27885     }
27886 }
27887 if (r == OK) {
27888     rip->i_gid = m_in.gid;
27889     rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
27890     rip->i_update |= CTIME;
27891     rip->i_dirt = DIRTY;
27892 }
27893
27894 put_inode(rip);
27895 return(r);
27896 }

27897 /*-----*
27898 *          do_umask
27899 *-----*/
27900 PUBLIC int do_umask()
27901 {
27902 /* Perform the umask(co_mode) system call. */
27903 register mode_t r;
27904
27905 r = ~fp->fp_umask;           /* set 'r' to complement of old mask */
27906 fp->fp_umask = ~(m_in.co_mode & RWX_MODES);
27907 return(r);                  /* return complement of old mask */
27908 }
27909 */

27910 /*-----*
27911 *          do_access
27912 *-----*/
27913 PUBLIC int do_access()
27914 {
27915 /* Perform the access(name, mode) system call. */
27916
27917     struct inode *rip;
27918     register int r;
27919
27920     /* First check to see if the mode is correct. */
27921     if ((m_in.mode & (R_OK | W_OK | X_OK)) != 0 && m_in.mode != F_OK)
27922         return(EINVAL);
27923
27924     /* Temporarily open the file whose access is to be checked. */
27925     if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
27926     if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
27927
27928     /* Now check the permissions. */
27929     r = forbidden(rip, (mode_t) m_in.mode);
27930     put_inode(rip);
27931     return(r);
27932 }
27933 }
```

```

27935 /*-----*
27936 *          .
27937 *-----* forbidden
27938 PUBLIC int forbidden(register struct inode *rip, mode_t access_desired)
27939 {
27940 /* Given a pointer to an inode, 'rip', and the access desired, determine
27941 * if the access is allowed, and if not why not. The routine looks up the
27942 * caller's uid in the 'fproc' table. If access is allowed, OK is returned
27943 * if it is forbidden, EACCES is returned.
27944 */
27945
27946 register struct inode *old_rip = rip;
27947 register struct super_block *sp;
27948 register mode_t bits, perm_bits;
27949 int r, shift, test_uid, test_gid, type;
27950
27951 if (rip->i_mount == I_MOUNT) /* The inode is mounted on. */
27952     for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++)
27953         if (sp->s_imount == rip) {
27954             rip = get_inode(sp->s_dev, ROOT_INODE);
27955             break;
27956         } /* if */
27957
27958 /* Isolate the relevant rwx bits from the mode. */
27959 bits = rip->i_mode;
27960 test_uid = (call_nr == ACCESS ? fp->fp_realluid : fp->fp_effuid);
27961 test_gid = (call_nr == ACCESS ? fp->fp_realgid : fp->fp_effgid);
27962 if (test_uid == SU_UID) {
27963     /* Grant read and write permission. Grant search permission for
27964      * directories. Grant execute permission (for non-directories) if
27965      * and only if one of the 'X' bits is set.
27966     */
27967     if ( (bits & I_TYPE) == I_DIRECTORY ||
27968         bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
27969         perm_bits = R_BIT | W_BIT | X_BIT;
27970     else
27971         perm_bits = R_BIT | W_BIT;
27972 } else {
27973     if (test_uid == rip->i_uid) shift = 6;           /* owner */
27974     else if (test_gid == rip->i_gid) shift = 3;       /* group */
27975     else shift = 0;                                /* other */
27976     perm_bits = (bits >> shift) & (R_BIT | W_BIT | X_BIT);
27977 }
27978
27979 /* If access desired is not a subset of what is allowed, it is refused. */
27980 r = OK;
27981 if ((perm_bits | access_desired) != perm_bits) r = EACCES;
27982
27983 /* Check to see if someone is trying to write on a file system that is
27984 * mounted read-only.
27985 */
27986 type = rip->i_mode & I_TYPE;
27987 if (r == OK)
27988     if (access_desired & W_BIT)
27989         r = read_only(rip);
27990
27991 if (rip != old_rip) put_inode(rip);
27992
27993 return(r);
27994 }

27996 /*-----*
27997 *          .
27998 *-----* read_only
27999 PUBLIC int read_only(ip)
28000 struct inode *ip;           /* ptr to inode whose file sys is to be cked */

```

```

28001 {
28002 /* Check to see if the file system on which the inode 'ip' resides is mounted
28003 * read only. If so, return EROFS, else return OK.
28004 */
28005
28006 register struct super_block *sp;
28007
28008 sp = ip->i_sp;
28009 return(sp->s_rd_only ? EROFS : OK);
28010 }

+++++servers/fs/dmap.c+++++
+++++servers/fs/dmap.c+++++

28100 /* This file contains the table with device <-> driver mappings. It also
28101 * contains some routines to dynamically add and/ or remove device drivers
28102 * or change mappings.
28103 */
28104
28105 #include "fs.h"
28106 #include "fproc.h"
28107 #include <string.h>
28108 #include <stdlib.h>
28109 #include <cctype.h>
28110 #include <unistd.h>
28111 #include <minix/com.h>
28112 #include "param.h"
28113
28114 /* Some devices may or may not be there in the next table. */
28115 #define DT(enable, opcl, io, driver, flags) \
28116 { (enable?(opcl):no_dev), (enable?(io):0), \
28117 (enable?(driver):0), (flags) },
28118 #define NC(x) (NR_CTRLRS >= (x))

28119 /* The order of the entries here determines the mapping between major device
28120 * numbers and tasks. The first entry (major device 0) is not used. The
28121 * next entry is major device 1, etc. Character and block devices can be
28122 * intermixed at random. The ordering determines the device numbers in /dev/.
28123 * Note that FS knows the device number of /dev/ram/ to load the RAM disk.
28124 * Also note that the major device numbers used in /dev/ are NOT the same as
28125 * the process numbers of the device drivers.
28126 */
28127 */
28128 /*
28129   Driver enabled      Open/Cls  I/O      Driver #    Flags Device  File
28130   -----      -----  -----      -----    ----- -----
28131 */
28132 struct dmap dmap[NR_DEVICES];                                /* actual map */
28133 PRIVATE struct dmap init_dmap[] = {
28134   DT(1, no_dev, 0, 0, 0)                                     /* 0 = not used */
28135   DT(1, gen_opcl, gen_io, MEM_PROC_NR, 0)                   /* 1 = /dev/mem */
28136   DT(0, no_dev, 0, 0, DMAP_MUTABLE)                          /* 2 = /dev/fd0 */
28137   DT(0, no_dev, 0, 0, DMAP_MUTABLE)                          /* 3 = /dev/c0 */
28138   DT(1, tty_opcl, gen_io, TTY_PROC_NR, 0)                  /* 4 = /dev/tty00 */
28139   DT(1, ctty_opcl, ctty_io, TTY_PROC_NR, 0)                /* 5 = /dev/tty */
28140   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                      /* 6 = /dev/lp */
28141   DT(1, no_dev, 0, 0, DMAP_MUTABLE)                          /* 7 = /dev/ip */
28142   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                      /* 8 = /dev/c1 */
28143   DT(0, 0, 0, 0, DMAP_MUTABLE)                            /* 9 = not used */
28144   DT(0, no_dev, 0, 0, DMAP_MUTABLE)                          /* 10 = /dev/c2 */
28145   DT(0, 0, 0, 0, DMAP_MUTABLE)                            /* 11 = not used */
28146   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                     /* 12 = /dev/c3 */
28147   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                     /* 13 = /dev/audio */
28148   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                     /* 14 = /dev/mixer */
28149   DT(1, gen_opcl, gen_io, LOG_PROC_NR, 0)                 /* 15 = /dev/klog */
28150   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                     /* 16 = /dev/random */
28151   DT(0, no_dev, 0, NONE, DMAP_MUTABLE)                     /* 17 = /dev/cmos */
28152 };

```

```

28153  *-----*
28154  *          do_devctl           *
28155  *-----*
28156  PUBLIC int do_devctl()
28157  {
28158      int result;
28159
28160      switch(m_in.ctl_req) {
28161          case DEV_MAP:
28162              /* Try to update device mapping. */
28163              result = map_driver(m_in.dev_nr, m_in.driver_nr, m_in.dev_style);
28164              break;
28165          case DEV_UNMAP:
28166              result = ENOSYS;
28167              break;
28168          default:
28169              result = EINVAL;
28170      }
28171      return(result);
28172  }

28175  *-----*
28176  *          map_driver           *
28177  *-----*
28178  PUBLIC int map_driver(major, proc_nr, style)
28179  int major;                      /* major number of the device */
28180  int proc_nr;                    /* process number of the driver */
28181  int style;                     /* style of the device */
28182  {
28183      /* Set a new device driver mapping in the dmap table. Given that correct
28184      * arguments are given, this only works if the entry is mutable and the
28185      * current driver is not busy.
28186      * Normal error codes are returned so that this function can be used from
28187      * a system call that tries to dynamically install a new driver.
28188      */
28189      struct dmap *dp;
28190
28191      /* Get pointer to device entry in the dmap table. */
28192      if (major >= NR_DEVICES) return(ENODEV);
28193      dp = &dmap[major];
28194
28195      /* See if updating the entry is allowed. */
28196      if (!(dp->dmap_flags & DMAP_MUTABLE)) return(EPERM);
28197      if (dp->dmap_flags & DMAP_BUSY) return(EBUSY);
28198
28199      /* Check process number of new driver. */
28200      if (!isokprocnr(proc_nr)) return(EINVAL);
28201
28202      /* Try to update the entry. */
28203      switch (style) {
28204          case STYLE_DEV:    dp->dmap_opcl = gen_opcl;      break;
28205          case STYLE_TTY:   dp->dmap_opcl = tty_opcl;      break;
28206          case STYLE_CLONE: dp->dmap_opcl = clone_opcl;    break;
28207          default:         return(EINVAL);
28208      }
28209      dp->dmap_io = gen_io;
28210      dp->dmap_driver = proc_nr;
28211      return(OK);
28212  }

28214  *-----*
28215  *          build_dmap           *
28216  *-----*
28217  PUBLIC void build_dmap()
28218  {
28219      /* Initialize the table with all device <-> driver mappings. Then, map

```

```

28220 * the boot driver to a controller and update the dmap table to that
28221 * selection. The boot driver and the controller it handles are set at
28222 * the boot monitor.
28223 */
28224 char driver[16];
28225 char *controller = "c##";
28226 int nr, major = -1;
28227 int i,s;
28228 struct dmap *dp;
28229
28230 /* Build table with device <-> driver mappings. */
28231 for (i=0; i<NR_DEVICES; i++) {
28232     dp = &dmap[i];
28233     if (i < sizeof(init_dmap)/sizeof(struct dmap) &&
28234         init_dmap[i].dmap_opcl != no_dev) { /* a preset driver */
28235         dp->dmap_opcl = init_dmap[i].dmap_opcl;
28236         dp->dmap_io = init_dmap[i].dmap_io;
28237         dp->dmap_driver = init_dmap[i].dmap_driver;
28238         dp->dmap_flags = init_dmap[i].dmap_flags;
28239     } else { /* no default */
28240         dp->dmap_opcl = no_dev;
28241         dp->dmap_io = 0;
28242         dp->dmap_driver = 0;
28243         dp->dmap_flags = DMAP_MUTABLE;
28244     }
28245 }
28246
28247 /* Get settings of 'controller' and 'driver' at the boot monitor. */
28248 if ((s = env_get_param("label", driver, sizeof(driver))) != OK)
28249     panic(FILE__, "couldn't get boot monitor parameter 'driver'", s);
28250 if ((s = env_get_param("controller", controller, sizeof(controller))) != OK)
28251     panic(FILE__, "couldn't get boot monitor parameter 'controller'", s);
28252
28253 /* Determine major number to map driver onto. */
28254 if (controller[0] == 'f' && controller[1] == 'd') {
28255     major = FLOPPY_MAJOR;
28256 }
28257 else if (controller[0] == 'c' && isdigit(controller[1])) {
28258     if ((nr = (unsigned) atoi(&controller[1])) > NR_CTRLRS)
28259         panic(FILE__, "monitor 'controller' maximum 'c#' is", NR_CTRLRS);
28260     major = CTRLR(nr);
28261 }
28262 else {
28263     panic(FILE__, "monitor 'controller' syntax is 'c#' or 'fd'", NO_NUM);
28264 }
28265
28266 /* Now try to set the actual mapping and report to the user. */
28267 if ((s=map_driver(major, DRVR_PROC_NR, STYLE_DEV)) != OK)
28268     panic(FILE__, "map_driver failed", s);
28269 printf("Boot medium driver: %s driver mapped onto controller %s.\n",
28270        driver, controller);
28271 }
```

```
+++++
servers/fs/device.c
+++++
```

```

28300 /* When a needed block is not in the cache, it must be fetched from the disk.
28301 * Special character files also require I/O. The routines for these are here.
28302 *
28303 * The entry points in this file are:
28304 *   dev_open:   FS opens a device
28305 *   dev_close:  FS closes a device
28306 *   dev_io:     FS does a read or write on a device
```

```

28307 * dev_status: FS processes callback request alert
28308 * gen_opcl: generic call to a task to perform an open/close
28309 * gen_io: generic call to a task to perform an I/O operation
28310 * no_dev: open/close processing for devices that don't exist
28311 * tty_opcl: perform tty-specific processing for open/close
28312 * ctty_opcl: perform controlling-tty-specific processing for open/close
28313 * ctty_io: perform controlling-tty-specific processing for I/O
28314 * do_ioctl: perform the IOCTL system call
28315 * do_setsid: perform the SETSID system call (FS side)
28316 */
28317
28318 #include "fs.h"
28319 #include <fcntl.h>
28320 #include <minix/callnr.h>
28321 #include <minix/com.h>
28322 #include "file.h"
28323 #include "fproc.h"
28324 #include "inode.h"
28325 #include "param.h"
28326
28327 #define ELEMENTS(a) (sizeof(a)/sizeof((a)[0]))
28328
28329 extern int dmap_size;
28330
28331 /*=====
28332 *          dev_open
28333 *=====
28334 PUBLIC int dev_open(dev, proc, flags)
28335 dev_t dev;           /* device to open */
28336 int proc;            /* process to open for */
28337 int flags;           /* mode bits and flags */
28338 {
28339     int major, r;
28340     struct dmap *dp;
28341
28342     /* Determine the major device number call the device class specific
28343      * open/close routine. (This is the only routine that must check the
28344      * device number for being in range. All others can trust this check.)
28345      */
28346     major = (dev >> MAJOR) & BYTE;
28347     if (major >= NR_DEVICES) major = 0;
28348     dp = &dmap[major];
28349     r = (*dp->dmap_opcl)(DEV_OPEN, dev, proc, flags);
28350     if (r == SUSPEND) panic(__FILE__, "suspend on open from", dp->dmap_driver);
28351     return(r);
28352 }
28353
28354 /*=====
28355 *          dev_close
28356 *=====
28357 PUBLIC void dev_close(dev)
28358 dev_t dev;           /* device to close */
28359 {
28360     (void) (*dmap[(dev >> MAJOR) & BYTE].dmap_opcl)(DEV_CLOSE, dev, 0, 0);
28361 }
28362
28363 /*=====
28364 *          dev_status
28365 *=====
28366 PUBLIC void dev_status(message *m)
28367 {
28368     message st;
28369     int d, get_more = 1;
28370
28371     for(d = 0; d < NR_DEVICES; d++)
28372         if (dmap[d].dmap_driver == m->m_source)
28373             break;
28374 }
```

```

28375     if (d >= NR_DEVICES)
28376         return;
28377
28378     do {
28379         int r;
28380         st.m_type = DEV_STATUS;
28381         if ((r=sendrec(m->m_source, &st)) != OK)
28382             panic(FILE, "couldn't sendrec for DEV_STATUS", r);
28383
28384         switch(st.m_type) {
28385             case DEV_REVIVE:
28386                 revive(st.REP_PROC_NR, st.REP_STATUS);
28387                 break;
28388             case DEV_IO_READY:
28389                 select_notified(d, st.DEV_MINOR, st.DEV_SEL_OPS);
28390                 break;
28391             default:
28392                 printf("FS: unrecognized rep %d to DEV_STATUS\n", st.m_type);
28393                 /* Fall through. */
28394             case DEV_NO_STATUS:
28395                 get_more = 0;
28396                 break;
28397         }
28398     } while(get_more);
28399
28400     return;
28401 }

28403 /*=====
28404 *          dev_io
28405 *=====
28406 PUBLIC int dev_io(op, dev, proc, buf, pos, bytes, flags)
28407 int op;           /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
28408 dev_t dev;        /* major-minor device number */
28409 int proc;         /* in whose address space is buf? */
28410 void *buf;        /* virtual address of the buffer */
28411 off_t pos;       /* byte position */
28412 int bytes;       /* how many bytes to transfer */
28413 int flags;        /* special flags, like O_NONBLOCK */
28414 {
28415 /* Read or write from a device. The parameter 'dev' tells which one. */
28416 struct dmap *dp;
28417 message dev_mess;
28418
28419 /* Determine task dmap. */
28420 dp = &dmap[(dev >> MAJOR) & BYTE];
28421
28422 /* Set up the message passed to task. */
28423 dev_mess.m_type = op;
28424 dev_mess.DEVICE = (dev >> MINOR) & BYTE;
28425 dev_mess.POSITION = pos;
28426 dev_mess.PROC_NR = proc;
28427 dev_mess.ADDRESS = buf;
28428 dev_mess.COUNT = bytes;
28429 dev_mess.TTY_FLAGS = flags;
28430
28431 /* Call the task. */
28432 (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
28433
28434 /* Task has completed. See if call completed. */
28435 if (dev_mess.REP_STATUS == SUSPEND) {
28436     if (flags & O_NONBLOCK) {
28437         /* Not supposed to block. */
28438         dev_mess.m_type = CANCEL;
28439         dev_mess.PROC_NR = proc;
28440         dev_mess.DEVICE = (dev >> MINOR) & BYTE;
28441         (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
28442         if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;

```

```

28443     } else {
28444         /* Suspend user. */
28445         suspend(dp->dmap_driver);
28446         return(SUSPEND);
28447     }
28448 }
28449 return(dev_mess.REP_STATUS);
28450 }

28452 /*=====
28453 *                         gen_opcl
28454 *=====
28455 PUBLIC int gen_opcl(op, dev, proc, flags)
28456 int op;                      /* operation, DEV_OPEN or DEV_CLOSE */
28457 dev_t dev;                   /* device to open or close */
28458 int proc;                    /* process to open/close for */
28459 int flags;                  /* mode bits and flags */
28460 {
28461 /* Called from the dmap struct in table.c on opens & closes of special files.*/
28462     struct dmap *dp;
28463     message dev_mess;
28464
28465     /* Determine task dmap. */
28466     dp = &dmap[(dev >> MAJOR) & BYTE];
28467
28468     dev_mess.m_type    = op;
28469     dev_mess.DEVICE    = (dev >> MINOR) & BYTE;
28470     dev_mess.PROC_NR   = proc;
28471     dev_mess.COUNT     = flags;
28472
28473     /* Call the task. */
28474     (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
28475
28476     return(dev_mess.REP_STATUS);
28477 }

28479 /*=====
28480 *                         tty_opcl
28481 *=====
28482 PUBLIC int tty_opcl(op, dev, proc, flags)
28483 int op;                      /* operation, DEV_OPEN or DEV_CLOSE */
28484 dev_t dev;                   /* device to open or close */
28485 int proc;                    /* process to open/close for */
28486 int flags;                  /* mode bits and flags */
28487 {
28488 /* This procedure is called from the dmap struct on tty open/close. */
28489
28490     int r;
28491     register struct fproc *rfp;
28492
28493     /* Add O_NOCTTY to the flags if this process is not a session leader, or
28494     * if it already has a controlling tty, or if it is someone elses
28495     * controlling tty.
28496     */
28497     if (!fp->fp_sesldr || fp->fp_tty != 0) {
28498         flags |= O_NOCTTY;
28499     } else {
28500         for (rfp = &fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
28501             if (rfp->fp_tty == dev) flags |= O_NOCTTY;
28502         }
28503     }
28504
28505     r = gen_opcl(op, dev, proc, flags);
28506
28507     /* Did this call make the tty the controlling tty? */
28508     if (r == 1) {
28509         fp->fp_tty = dev;

```

```

28510         r = OK;
28511     }
28512     return(r);
28513 }

28515 /*=====
28516 *          ctty_opcl
28517 *=====
28518 PUBLIC int ctty_opcl(op, dev, proc, flags)
28519 int op;                                /* operation, DEV_OPEN or DEV_CLOSE */
28520 dev_t dev;                            /* device to open or close */
28521 int proc;                             /* process to open/close for */
28522 int flags;                            /* mode bits and flags */
28523 {
28524 /* This procedure is called from the dmap struct in table.c on opening/closing
   * /dev/tty, the magic device that translates to the controlling tty.
28525 */
28526
28527     return(fp->fp_tty == 0 ? ENXIO : OK);
28528 }
28529 */

28531 /*=====
28532 *          do_setsid
28533 *=====
28534 PUBLIC int do_setsid()
28535 {
28536 /* Perform the FS side of the SETSID call, i.e. get rid of the controlling
   * terminal of a process, and make the process a session leader.
28537 */
28538
28539 register struct fproc *rfp;
28540
28541 /* Only MM may do the SETSID call directly. */
28542 if (who != PM_PROC_NR) return(ENOSYS);
28543
28544 /* Make the process a session leader with no controlling tty. */
28545 rfp = &fproc[m_in.slot1];
28546 rfp->fp_sesldr = TRUE;
28547 rfp->fp_tty = 0;
28548 return(OK);
28549 }

28551 /*=====
28552 *          do_ioctl
28553 *=====
28554 PUBLIC int do_ioctl()
28555 {
28556 /* Perform the ioctl(ls_fd, request, argx) system call (uses m2 fmt). */
28557
28558 struct filp *f;
28559 register struct inode *rip;
28560 dev_t dev;
28561
28562 if ( (f = get_filp(m_in.ls_fd)) == NIL_FILP) return(err_code);
28563 rip = f->filp_ino; /* get inode pointer */
28564 if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
28565     && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
28566 dev = (dev_t) rip->i_zone[0];
28567
28568 return(dev_io(DEV_IOCTL, dev, who, m_in.ADDRESS, 0L,
28569             m_in.REQUEST, f->filp_flags));
28570 }

28572 /*=====
28573 *          gen_io
28574 *=====
28575 PUBLIC void gen_io(task_nr, mess_ptr)
28576 int task_nr; /* which task to call */

```

```

28577 message *mess_ptr;           /* pointer to message for task */
28578 {
28579 /* All file system I/O ultimately comes down to I/O on major/minor device
28580 * pairs. These lead to calls on the following routines via the dmap table.
28581 */
28582
28583     int r, proc_nr;
28584     message local_m;
28585
28586     proc_nr = mess_ptr->PROC_NR;
28587     if (!isokprocnr(proc_nr)) {
28588         printf("FS: warning, got illegal process number (%d) from %d\n",
28589             mess_ptr->PROC_NR, mess_ptr->m_source);
28590         return;
28591     }
28592
28593     while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
28594         /* sendrec() failed to avoid deadlock. The task 'task_nr' is
28595         * trying to send a REVIVE message for an earlier request.
28596         * Handle it and go try again.
28597         */
28598         if ((r = receive(task_nr, &local_m)) != OK) {
28599             break;
28600         }
28601
28602         /* If we're trying to send a cancel message to a task which has just
28603         * sent a completion reply, ignore the reply and abort the cancel
28604         * request. The caller will do the revive for the process.
28605         */
28606         if (mess_ptr->m_type == CANCEL && local_m.REP_PROC_NR == proc_nr) {
28607             return;
28608         }
28609
28610         /* Otherwise it should be a REVIVE. */
28611         if (local_m.m_type != REVIVE) {
28612             printf(
28613                 "fs: strange device reply from %d, type = %d, proc = %d (1)\n",
28614                     local_m.m_source,
28615                     local_m.m_type, local_m.REP_PROC_NR);
28616             continue;
28617         }
28618
28619         revive(local_m.REP_PROC_NR, local_m.REP_STATUS);
28620     }
28621
28622     /* The message received may be a reply to this call, or a REVIVE for some
28623     * other process.
28624     */
28625     for (;;) {
28626         if (r != OK) {
28627             if (r == EDEADDST) return; /* give up */
28628             else panic(__FILE__, "call_task: can't send/receive", r);
28629         }
28630
28631         /* Did the process we did the sendrec() for get a result? */
28632         if (mess_ptr->REP_PROC_NR == proc_nr) {
28633             break;
28634         } else if (mess_ptr->m_type == REVIVE) {
28635             /* Otherwise it should be a REVIVE. */
28636             revive(mess_ptr->REP_PROC_NR, mess_ptr->REP_STATUS);
28637         } else {
28638             printf(
28639                 "fs: strange device reply from %d, type = %d, proc = %d (2)\n",
28640                     mess_ptr->m_source,
28641                     mess_ptr->m_type, mess_ptr->REP_PROC_NR);
28642         }
28643     }
28644 }
```

```

28643      }
28644      r = receive(task_nr, mess_ptr);
28645    }
28646  }
28647 }

28649 /*=====
28650 *          ctty_io
28651 *=====
28652 PUBLIC void ctty_io(task_nr, mess_ptr)
28653 int task_nr;           /* not used - for compatibility with dmap_t */
28654 message *mess_ptr;     /* pointer to message for task */
28655 {
28656 /* This routine is only called for one device, namely /dev/tty. Its job
28657 * is to change the message to use the controlling terminal, instead of the
28658 * major/minor pair for /dev/tty itself.
28659 */
28660
28661 struct dmap *dp;
28662
28663 if (fp->fp_tty == 0) {
28664     /* No controlling tty present anymore, return an I/O error. */
28665     mess_ptr->REP_STATUS = EIO;
28666 } else {
28667     /* Substitute the controlling terminal device. */
28668     dp = &dmap[(fp->fp_tty >> MAJOR) & BYTE];
28669     mess_ptr->DEVICE = (fp->fp_tty >> MINOR) & BYTE;
28670     (*dp->dmap_io)(dp->dmap_driver, mess_ptr);
28671 }
28672 }

28674 /*=====
28675 *          no_dev
28676 *=====
28677 PUBLIC int no_dev(op, dev, proc, flags)
28678 int op;                  /* operation, DEV_OPEN or DEV_CLOSE */
28679 dev_t dev;                /* device to open or close */
28680 int proc;                 /* process to open/close for */
28681 int flags;                /* mode bits and flags */
28682 {
28683 /* Called when opening a nonexistent device. */
28684
28685     return(ENODEV);
28686 }

28688 /*=====
28689 *          clone_opcl
28690 *=====
28691 PUBLIC int clone_opcl(op, dev, proc, flags)
28692 int op;                  /* operation, DEV_OPEN or DEV_CLOSE */
28693 dev_t dev;                /* device to open or close */
28694 int proc;                 /* process to open/close for */
28695 int flags;                /* mode bits and flags */
28696 {
28697 /* Some devices need special processing upon open. Such a device is "cloned",
28698 * i.e. on a successful open it is replaced by a new device with a new unique
28699 * minor device number. This new device number identifies a new object (such
28700 * as a new network connection) that has been allocated within a task.
28701 */
28702     struct dmap *dp;
28703     int minor;
28704     message dev_mess;
28705
28706 /* Determine task dmap. */
28707     dp = &dmap[(dev >> MAJOR) & BYTE];
28708     minor = (dev >> MINOR) & BYTE;
28709

```

```

28710    dev_mess.m_type   = op;
28711    dev_mess.DEVICE   = minor;
28712    dev_mess.PROC_NR  = proc;
28713    dev_mess.COUNT    = flags;
28714
28715    /* Call the task. */
28716    (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
28717
28718    if (op == DEV_OPEN && dev_mess.REP_STATUS >= 0) {
28719        if (dev_mess.REP_STATUS != minor) {
28720            /* A new minor device number has been returned. Create a
28721               * temporary device file to hold it.
28722               */
28723            struct inode *ip;
28724
28725            /* Device number of the new device. */
28726            dev = (dev & ~(BYTE << MINOR)) | (dev_mess.REP_STATUS << MINOR);
28727
28728            ip = alloc_inode(root_dev, ALL_MODES | I_CHAR_SPECIAL);
28729            if (ip == NIL_INODE) {
28730                /* Oops, that didn't work. Undo open. */
28731                (void) clone_opcl(DEV_CLOSE, dev, proc, 0);
28732                return(err_code);
28733            }
28734            ip->i_zone[0] = dev;
28735
28736            put_inode(fp->fp_filp[m_in.fd]->filp_ino);
28737            fp->fp_filp[m_in.fd]->filp_ino = ip;
28738        }
28739        dev_mess.REP_STATUS = OK;
28740    }
28741    return(dev_mess.REP_STATUS);
28742 }

```

```

+++++
servers/fs/time.c
+++++
28800 /* This file takes care of those system calls that deal with time.
28801 */
28802 /* The entry points into this file are
28803 * do_utime:           perform the UTIME system call
28804 * do_stime:           PM informs FS about STIME system call
28805 */
28806
28807 #include "fs.h"
28808 #include <minix/callnr.h>
28809 #include <minix/com.h>
28810 #include "file.h"
28811 #include "fproc.h"
28812 #include "inode.h"
28813 #include "param.h"
28814
28815 /*=====
28816 *          do_utime
28817 *=====
28818 PUBLIC int do_utime()
28819 {
28820     /* Perform the utime(name, timep) system call. */
28821
28822     register struct inode *rip;
28823     register int len, r;
28824

```

```
28825 /* Adjust for case of 'timep' being NULL;
28826     * utime_strlen then holds the actual size: strlen(name)+1.
28827     */
28828 len = m_in.utime_length;
28829 if (len == 0) len = m_in.utime_strlen;
28830
28831 /* Temporarily open the file. */
28832 if (fetch_name(m_in.utime_file, len, M1) != OK) return(err_code);
28833 if ((rip = eat_path(user_path)) == NIL_INODE) return(err_code);
28834
28835 /* Only the owner of a file or the super_user can change its time. */
28836 r = OK;
28837 if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;
28838 if (m_in.utime_length == 0 && r != OK) r = forbidden(rip, W_BIT);
28839 if (read_only(rip) != OK) r = EROFS; /* not even su can touch if R/O */
28840 if (r == OK) {
28841     if (m_in.utime_length == 0) {
28842         rip->i_atime = clock_time();
28843         rip->i_mtime = rip->i_atime;
28844     } else {
28845         rip->i_atime = m_in.utime_actime;
28846         rip->i_mtime = m_in.utime_modtime;
28847     }
28848     rip->i_update = CTIME; /* discard any stale ATIME and MTIME flags */
28849     rip->i_dirt = DIRTY;
28850 }
28851
28852 put_inode(rip);
28853 return(r);
28854 }

28855 /*=====
28856      *                         do_stime
28857      *=====
28858 PUBLIC int do_stime()
28859 {
28860 /* Perform the stime(tp) system call. */
28861 boottime = (long) m_in.pm_stime;
28862 return(OK);
28863 }
```

## 附录C 文件索引

### include 目录

00000 include/ansi.h  
00200 include/errno.h  
00900 include/fcntl.h  
00100 include/limits.h  
00700 include/signal.h  
00600 include/string.h  
01000 include/termios.h  
01300 include/timers.h  
00400 include/unistd.h  
04400 include(ibm/interrupt.h  
04300 include(ibm/portio.h  
04500 include(ibm/ports.h  
03500 include/minix/callnr.h  
03600 include/minix/com.h  
02300 include/minix/config.h  
02600 include/minix/const.h  
04100 include/minix/devio.h  
04200 include/minix/dmap.h  
02200 include/minix/ioctl.h  
03000 include/minix/ipc.h  
02500 include/minix/sys\_config.h  
03200 include/minix/syslib.h  
03400 include/minix/sysutil.h  
02800 include/minix/type.h  
01800 include/sys/dir.h  
02100 include/sys/ioc\_disk.h  
02000 include/sys/ioctl.h  
01600 include/sys/sigcontext.h  
01700 include/sys/stat.h  
01400 include/sys/types.h  
01900 include/sys/wait.h

### 驱动

10800 drivers/drivers.h

12100 drivers/at\_wini/at\_wini.c  
12000 drivers/at\_wini/at\_wini.h  
11000 drivers/libdriver driver.c  
10800 drivers/libdriver driver.h  
11400 drivers/libdriver/drplib.c  
10900 drivers/libdriver/drplib.h  
11600 drivers/memory/memory.c  
15900 drivers/tty/console.c  
15200 drivers/tty/keyboard.c  
13600 drivers/tty/tty.c  
13400 drivers/tty/tty.h

### 内核

10400 kernel/clock.c  
04700 kernel/config.h  
04800 kernel/const.h  
08000 kernel/exception.c  
05300 kernel/glo.h  
08100 kernel/i8259.c  
05400 kernel/ ipc.h  
04600 kernel/kernel.h  
08700 kernel/klib.s  
08800 kernel/klib386.s  
07100 kernel/main.c  
06200 kernel/mpx.s  
06300 kernel/mpx386.s  
05700 kernel/priv.h  
07400 kernel/proc.c  
05500 kernel/proc.h  
08300 kernel/protect.c  
05800 kernel/protect.h  
05100 kernel/proto.h  
05600 kernel/sconst.h  
06900 kernel/start.c  
09700 kernel/system.c

09600 kernel/system.h	27800 servers/fs/protect.c	
10300 kernel/system/do_exec.c	21200 servers/fs/proto.h	
10200 kernel/system/do_setalarm.c	25000 servers/fs/read.c	
06000 kernel/table.c	27500 servers/fs/stadir.c	
04900 kernel/type.h	23300 servers/fs/super.c	
09400 kernel/utility.c	22100 servers/fs/super.h	
<b>文件系统</b>		
21600 servers/fs/buf.h	22200 servers/fs/table.c	
22400 servers/fs/cache.c	28800 servers/fs/time.c	
21000 servers/fs/const.h	21100 servers/fs/type.h	
28300 servers/fs/device.c	25600 servers/fs/write.c	
28100 servers/fs/dmap.c	<b>进程管理器</b>	
21700 servers/fs/file.h	19300 servers/pm/break.c	
23700 servers/fs/filedes.c	17100 servers/pm/const.h	
21500 servers/fs/fproc.h	18700 servers/pm/exec.c	
20900 servers/fs/fs.h	18400 servers/pm/forkexit.c	
21400 servers/fs/glo.h	20400 servers/pm/getset.c	
22900 servers/fs/inode.c	17500 servers/pm/glo.h	
21900 servers/fs/inode.h	18000 servers/pm/main.c	
27000 servers/fs/link.c	20500 servers/pm/misc.c	
23800 servers/fs/lock.c	17600 servers/pm/mproc.h	
21800 servers/fs/lock.h	17700 servers/pm/param.h	
24000 servers/fs/main.c	17000 servers/pm/pm.h	
26700 servers/fs/mount.c	17300 servers/pm/proto.h	
24500 servers/fs/open.c	19500 servers/pm/signal.c	
22000 servers/fs/param.h	17800 servers/pm/table.c	
26300 servers/fs/path.c	20300 servers/pm/time.c	
25900 servers/fs/pipe.c	20200 servers/pm/timers.c	
	17200 servers/pm/type.h	

封面

书名

版权

前言

目录

附录A 安装MINIX 3

附录B MINIX源码

附录C 文件索引