# Contents

1 Introduction to Weenix ........................................... 5
   1.1 Why Unix? .................................................. 5
   1.2 Simplifications ........................................... 5
   1.3 The Language Issue ....................................... 6

2 Processes and Threads ........................................... 7
   2.1 Weenix Processes ......................................... 7
   2.2 Threads .................................................. 8
      2.2.1 Synchronization .................................... 8
      2.2.2 Scheduling Threads ................................ 8

3 Virtual File System ............................................. 11
   3.1 A Common Interface ...................................... 11
   3.2 A Flexible Hierarchy ..................................... 11
   3.3 'Virtual' File System .................................... 12
      3.3.1 Object Oriented Design in C ......................... 12
      3.3.2 vfs_type_t .......................................... 13
      3.3.3 vfs_t ............................................. 13
      3.3.4 vnode_t ........................................... 13
      3.3.5 file_t ............................................ 14
   3.4 Reference counting ....................................... 15

A Debugging under Weenix ........................................ 17
   A.1 dbx .......................................................... 17
      A.1.1 CS169 .dbxrc ........................................ 17
   A.2 Debug modes .............................................. 18
      A.2.1 dbg(mode, stuffToPassToPrinf) ..................... 19
      A.2.2 Your DBG environment variable ....................... 19
      A.2.3 Adding your own modes ................................ 19
   A.3 Random kernel macros .................................... 19
      A.3.1 Debugging Macros .................................... 19
      A.3.2 VFS Macros ......................................... 19
      A.3.3 Core Map Macros .................................... 20
      A.3.4 List Macros ......................................... 20
      A.3.5 Virtual Memory Macros .............................. 20
   A.4 Kernel Datastructure Printing Functions .................... 21
      A.4.1 Installing ........................................... 21
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.4.2 Using</td>
<td>21</td>
</tr>
<tr>
<td>A.4.3 List of functions</td>
<td>21</td>
</tr>
<tr>
<td><strong>A Linux Kernel Coding Style</strong></td>
<td></td>
</tr>
<tr>
<td>A.1 Indentation</td>
<td>23</td>
</tr>
<tr>
<td>A.2 Placing Braces</td>
<td>23</td>
</tr>
<tr>
<td>A.3 Naming</td>
<td>24</td>
</tr>
<tr>
<td>A.4 Functions</td>
<td>25</td>
</tr>
<tr>
<td>A.5 Commenting</td>
<td>25</td>
</tr>
<tr>
<td>A.6 You’ve made a mess of it</td>
<td>26</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction to Weenix

Weenix is a single-user, preemptively multitasking, virtual memory operating system intended for educational use. Although Weenix does not adhere to any formal standards, it exports a unix-like\(^1\) interface to users. Its implementation, while simple, is not a caricature. Ideas have been borrowed from Linux, BSD, and the descendants of AT&T’s System V Release 4.

As provided by the TA staff, Weenix is incomplete. You will write the missing portions of it over the course of the semester.

This document should serve as both orientation and reference. You do not need to understand all of the material here in one sitting; we expect that you will return to it over the course of the semester, filling in gaps in your understanding as needed.

1.1 Why Unix?

Weenix is an undergrown member of the unix family of operating systems. There are many reasons, sensible and otherwise, for liking or disliking unix; for the purposes of this document, we don’t care. Whether you love unix or you hate it, it’s a great entrypoint to a study of operating systems.

Since unix is a general-purpose system, it addresses all of the major issues of operating systems. In spite of this generality, a simple implementation is possible: in fact, a student can gain a comprehensive understanding of a no-frills unix implementation in one semester of study.

Unix is intrinsically interesting to a student of operating systems. The basic abstractions of unix are intact today, more than twenty-five years after its birth. Unix’s fundamental design has proved flexible enough to accommodate many unforeseen changes in computing—networks, virtual memory, graphical displays, multiprocessors, etc. Unix is a uniquely successful operating system; understanding it is part of basic OS literacy.

1.2 Simplifications

Since Weenix is more educational than useful, it lacks many features expected of a real unix. Some of what’s missing in Weenix:

- signals;

\(^1\)When this document refers to "unix," you can assume that it is referring to all operating systems that are commonly called "unix." We’ll be more specific when necessary.
CHAPTER 1. INTRODUCTION TO WEENIX

- pipes;
- multiple users;
- multiprocessor support;
- paging to backing store;
- etc.

We felt that most of these features’ educational benefits did not justify the difficulty of implementing them. Paging to backing store and multiprocessor support, while interesting and worthwhile, are beyond the scope of this course.

1.3 The Language Issue

Weenix is written in the common subset of C and C++. This means that students can use the language of their choice in implementing assignments. It also means that the code looks a lot more like C than C++.

Writing the kernel in C eased implementation; since all the unix kernels that Weenix’s implementors were familiar with were in C, translating designs into C++ would have required some effort. The choice of C also reflects usual practice; since the support code is written in the same language and style as most other unix kernels (see Appendix A), the source to most other unix kernels will seem familiar to you after a thorough study of Weenix.

Of course, if you feel strongly about object-orientation, and find the support code as written unacceptable, remember the CS 169 motto: "It’s not our code; it’s your code." You are free to alter the support code in any way you find useful. If you would like to change the code to better suit the C++ way of thinking, please do so.

\footnote{Thomas W. Doppelner, unpublished}
Chapter 2

Processes and Threads

Like all unixes, Weenix provides the “process” as a fundamental abstraction. The traditional definition of a process is, “a program in execution.” A process is the operating system’s internal representation of the state of an executing program.

One aspect of the state of an executing program is the memory that the program is using. It is critical that no program be able to arbitrarily change the contents of another program’s memory; a buggy or malicious program could then cause other programs to malfunction. It is the operating system’s responsibility to use the memory management hardware to keep processes out of one another’s way. This babysitting is easier said than done; see chapter ?? for the details.

Another part of an executing program’s state is the contents of its registers on the CPU it is using. We abstract this state into a “thread.” A thread is sometimes defined as an execution vehicle. From the operating system’s perspective, a thread is an entity which would like to use the CPU. The OS gives each user-level thread the illusion that it has a CPU all to itself. Weenix must hide the ugly realities (e.g., interrupts, other threads that need to use the same physical CPU) that might disturb this illusion.

2.1 Weenix Processes

The proc_t type, defined in proc.h, is the structure representing a process. The kernel keeps a large global table of proc_t’s. This table, named procs, is defined in proc.c. Its size limits the number of processes that may be active at one time on a Weenix system.

Some of the members of proc_t are only pertinent to Weenix’s memory management; we will wait until Chapter ?? to cover those. The rest of the members are:

- p_threads – a list of the active threads belonging to this process.
- p_pid – a small integer uniquely identifying this process, conventionally known in UNIX as the “pid,” for “process ID.” As implemented in Weenix, this is the index of the process’s entry in procs.
- p_pproc – a pointer to the parent of this process; i.e., the process that called fork(t) to create this process.
- p_status – this process’s return value from main(,) or argument to exit(,) if the process has exited.
• \texttt{p\_state} – one of \texttt{PROC\_NONE}, \texttt{PROC\_RUNNING}, or \texttt{PROC\_DEAD}. See the comments accompanying these constants’ definitions in \texttt{proc.h}.

• \texttt{p\_nchild} – number of running child processes of this process. This is used to implement the \texttt{wait(s)ystem} call.

• \texttt{p\_files} – an array of open files. See 3.

When the kernel creates a new process, it searches \texttt{procs} for an entry whose \texttt{p\_state} field is \texttt{PROC\_NONE}.

### 2.2 Threads

The Weenix kernel is prepared for the presence of multiple threads within a single process; however, this functionality is not visible at user-level. Weenix makes the distinction between thread and process only for structural discipline.

The kernel thread structure, \texttt{kthread\_t}, defined in \texttt{kthread.h}, maintains all of the state of a running kernel thread. This includes the process to which this thread belongs, the thread’s simulator context, its exit value, its kernel and user stacks, and its errno value (the error return of the most recent system call).

Notice that threads have separate kernel and user stacks. The kernel stack is in physical memory, whereas the user stack is a pointer in the process’s virtual address space. When a thread running at user-level takes an interrupt or trap, it begins executing on the thread’s kernel stack.

#### 2.2.1 Synchronization

Your kernel will need some synchronization primitives in order to control access to resources; for example, only one thread may be writing to a given terminal at a time. You can use any set of synchronization primitives you like; the TAs chose to use mutexes and condition variables, out of familiarity. The CS 167 lectures on synchronization should help you implement your chosen primitives efficiently and correctly.

#### 2.2.2 Scheduling Threads

Each thread has the illusion that it has the processor to itself, but the kernel doesn’t have this luxury. When there are multiple threads wishing to use the CPU, we must choose only one of them to run, in such a way that every runnable thread eventually gets to run. The part of the operating system that decides which thread runs at any given time is called the scheduler.

The scheduler that we ask you to write for Weenix is very simple: we maintain a global queue, \texttt{runq}, that contains all runnable threads. When we enter the scheduler, by calling \texttt{swtch(,)} we pick a runnable thread from the head of the queue and begin executing it. Whenever any thread becomes runnable, it is placed on the tail of the run queue. This simple algorithm has an obvious flaw: a buggy or malicious program can monopolize the cpu. A simple program that spins in an infinite loop will bring the system to a halt.

The TA scheduler also makes use of a technique sometimes known as time-slicing, or preemption. We maintain a per-thread counter, \texttt{kt\_quantum}, which represents the kernel’s rough

\(^{1}\)This is certainly an easy area for improvement. In addition, you may find it useful to add additional fields to the \texttt{proc} structure to implement \texttt{wait(2)} and friends.
idea of how much CPU time the corresponding thread is entitled to. When the kernel receives a
clock interrupt, we decrement $kt_{\text{quantum}}$. Then, when exiting the kernel, if the current thread's
$kt_{\text{quantum}}$ is zero, and it was a user-mode context that we interrupted\(^2\), we yield the CPU,
preepting the currently running user thread. Exercise for the reader: when should we reset
$kt_{\text{quantum}}$?

The Weenix scheduler is surprisingly effective, considering how little effort its implementation
requires. However, the modest scheduler we describe here would be inadequate for any real kernel;
even with preemption, it is too easy for a user program to monopolize the CPU. In real operating
systems, the scheduler is a very complex subsystem, and scheduling considerations permeate the
design of the entire kernel. The scheduler is a good opportunity for extra-credit extensions to
the base Weenix spec. We refer you to the 167 slides on scheduling if you're looking for ideas.

\(^2\)Why can't we preempt a thread running in kernel mode? Weenix expects invocations of the kernel to be
atomic, unless they block. That is, we implicitly use the fact that we are in the kernel to synchronize a lot of
code. More modern UNIXes (e.g., Solaris and IRIX) don't have this limitation; adding explicit synchronization
to all of the implicitly synchronized code in a UNIX kernel is a tremendous effort. So, like Linux, Weenix takes
the easy way out and refuses to preempt threads running in kernel-mode. Besides, some would argue, if you're
spending so much time in the kernel that it's important to be able to preempt it, your kernel is too slow. If you're
interested in some of the things that might have to be done to Weenix to implement kernel-level preemption, talk
to some of your TAs.
Chapter 3

Virtual File System

3.1 A Common Interface

The Virtual File System (VFS) provides a common interface between the operating system kernel and the various file systems. The VFS interface allows one to add many different types of filesystems to one’s kernel and access them through the same UNIX-style interface: one can access one’s MS-DOS files via the vfat filesystem just as easily as one would access various device drivers through the dev filesystem, kernel internal information through the proc filesystem or standard on disk files through the SysV filesystem. For example, writing to a 'file':

```
$ cat foo.txt > /home/bar.txt
$ cat foo.txt > /dev/tty0
$ cat foo.txt > /proc/123/mem
```

All of these commands look very similar, but their effect is vastly different. Command 1 writes the contents of foo.txt into a file on disk via the s5fs filesystem. Command 2 writes the contents to a terminal via the device filesystem (much like write(2) would do). Command 3 writes the contents of foo.txt into the address space of process 123.

3.2 A Flexible Hierarchy

The virtual filesystem also allows for a flexible hierarchy of filesystems. The top most filesystem is referred to as the root filesystem and is found at '/'. Typically in Weenix this is your standard disk-based s5fs filesystem. Other filesystems can then be mounted on nodes of the root filesystem. What mounting accomplishes is forwarding requests under a specific directory path along to the appropriate filesystem.

An example with our old favorite\(^1\) Weenix binary 'ls':

```
$ ls
```

Being in the root (s5fs) filesystem 'ls' looks at various on disk structures to tell us the contents of this directory.

\(^1\)In past years, having a kernel that could successfully handle three concurrent calls to 'ls' from three different shells meant getting an 'A'. Nowadays you must run nasty programs like 'stress' and 's5fstest' while chewing gum and reciting the alphabet backwards before your mentor TA will even let you show him/her your kernel.
CHAPTER 3. VIRTUAL FILE SYSTEM

$ cd /dev

We start in the root filesystem and look for the /dev directory. We find it and discover that it is in fact a mount point for a whole new filesystem (the device filesystem).

/dev$ ls

Since we are now in the device filesystem the call to 'ls' prints a list of devices registered to the system.

### 3.3 'Virtual' File System

As can be inferred by the description thus far, the main power of the VFS is its polymorphic design. Generic calls to the VFS such as `lookup()` are implemented on a per filesystem basis, as are calls to files like `read()` and `write()`. The fact that these terms probably remind you of cs15 is no coincidence. VFS is definitely designed with object oriented principals in mind. There are four objects central to understanding the VFS: The `vfs_type_t`, the `vfs_t`, the `vnode_t` and the `file_t`.

#### 3.3.1 Object Oriented Design in C

Before we explain how this system works, there are a few implementation details that need to be addressed as to how we implement 'objects' in C.

---

2the function that powers `chdir(2)` among other things
3.3. 'VIRTUAL' FILE SYSTEM

Construction

Each object needs to be constructed according to its specific subclass instance. Thus for each filesystem, there are routines that create filesystem specific versions of each object. The naming convention we use in Weenix is `read_XXX`. Additionally some objects have the corresponding destructor `delete_XXX`. Construction does the expected thing with data members, initializing them to some well defined value.

Virtual Functions

Virtual functions are debatably what gives OOD its power. We implement them in Weenix via a struct of function pointers. Every virtual function has a pointer of its type in the `XXX_ops` struct of its object. Upon creation, pointers to the actual subclass implementation of these functions are assigned to this struct. One can then call these functions through the pointers, and voila! Polymorphism.

Subclass specific data

Some objects have subclass specific data associated with them. This is implemented by using a union. The union consists of one struct per subclass that has subclass specific data. Each struct then contains the data needed by that subclass. As in traditional OOP, one must know that the object is a specific type before utilizing its subclass specific data.

3.3.2 `vfs_type_t`

The `vfs_type_t` is a simple object created when one registers a filesystem with the VFS. It consists solely of a link on a list, the name of the filesystem, and a function pointer to the constructor for this filesystem’s version of a `vfs_t`.

3.3.3 `vfs_t`

A `vfs_t` represents a filesystem. Each filesystem in use on the system has one of these structures associated with it. A `vfs_t` contains information such as where the filesystem is mounted, the root of this filesystem, a device associated with this filesystem and additional information needed by specific filesystems. It also defines a constructor and destructor for this filesystem’s version of a `vnode_t`.

3.3.4 `vnode_t`

A `vnode_t` represents the abstraction of a file. As we now know, a file is not necessary a chunk of bytes on a disk, but simply anything that supports the interface required of a file. A `vnode_t` contains the following:

- information on what filesystem it belongs to
- any filesystems for which it is a mount point so that it can forward requests onto the root vnode of that filesystem

\(^3\)tc might disagree claiming that the true power of C++ is its protected abstract virtual base pure virtual private destructors.

\(^4\)This is not to be confused with either ‘/’ (the root of the VFS) nor is it the same thing as where the filesystem is mounted. See above figure if this is unclear.
• a vnode number which uniquely identifies it among its specific filesystem

• a type field specifying what 'type' of file it is. Current choices include REGULAR, DIRECTORY or BLOCK

• subclass specific data

In addition the following functions can be called on a vnode:

• create(-) called on a directory to create a vnode within

• lookup(-) called on a directory to find a vnode within

• link/unlink(-) called on a directory to link/unlink a vnode within

• mkdir/rmdir(-) called on a directory to mkdir/rmdir within

• getpage(-) used to bring in a page of memory from the appropriate place

Lastly, the vnode’s array of function pointers also contains a struct with all of a file’s function pointers. This is done in the interest of space efficiency. There are more file’s than vnode’s and every file of a particular vnode has the same set of functions associated with it.

3.3.5 file

A file represents an entry in the system file table. A file consists of a reference to a vnode, a variable indicating its open mode (read, write, readwrite), and a variable indicating its position within the file. Additionally it has the following functions associated with it: lseek - for changing position within the file, read, write and readdir - read for a directory.

---

5 this will become clearer once you get to VM
6 see twd’s lecture: Introduction to Unix
3.4. REFERENCE COUNTING

Both vnode_t’s and file_t’s have reference counts associated with them. They are modified using vget/fget to increment the count and vput/fput to decrement the count. The need for these counts arises from the fact that both vnode_t’s and file_t’s can be referenced by multiple things (vnode_t’s by multiple file_t’s, and file_t’s by multiple file descriptors in your proc_t). When one is done with a vnode_t or a file_t one would like to dispose of it. However if someone somewhere else is referencing it, that would of course be a BadThing(TM). The solution then is to up the reference count whenever one is using one of these objects, and down the reference count when one is done. If once you are done, the reference count equals 0, you can then safely delete the object without fear.
Appendix A

Debugging under Weenix

Debugging an operating system is always a challenge. Weenix is no exception. This document describes the hodgepodge of tools you can use to facilitate your debugging efforts.

A.1 dbx

Dbx is your friend. If you don’t already know and love her, you should start to, and soon.

A.1.1 CS169 .dbxrc

Provided in /course/cs167/pub/ this file is essential to your debugging efforts under Weenix. It provides a ton of time saving functions as well as some essential functions needed to use dbx to debug Weenix. To use this .dbxrc you can either copy it to your home directory, or copy it to wherever your Weenix code resides. By default dbx first tries to use the .dbxrc in the directory where it was invoked from, then looks for a .dbxrc in your home directory.

sbt

This function starts up Weenix. To begin a debug session, you should run 'dbx simrun' then 'sbt'. This causes your 'kernel.so' library to become loaded. Because of this, you cannot set breakpoints in your code before calling sbt.

hex int

Prints out a number in hexadecimal

bit int

Prints out a number in binary

print_pte pte

Prints out a decoded page table entry
framenum virtual address
Prints out the frame number of a virtual address

simtlb
Prints contents of the simulators TLB

simregs
Prints contents of the simulators registers

maps proc_t*
Prints the memory mappings of a process (current process if no process is specified)

thrs proc_t*
Prints the threads of a process (current process if no process is specified)

bufs
Prints the system buffer list in LRU order

cmap addr
Prints the core map entry for a physical page

print_vnode vnode_t*
Prints a vnode

pp expression
Print pointer - Prints expression as symbolic address in hex value

list_foreach list_t* type member ksh-code
Do something to each element of a list (see list.h)

A.2 Debug modes
Weenix has advanced support for printf use in debugging. When specifying a debugging printf, you can attach modes to the message. What the mode does is classify what portion of the operating system this printf is pertaining to. You can then easily turn on only those modes you are interested in seeing, and turn them off when you are done. This is especially useful when presenting your kernel to the TAs.
A.3. RANDOM KERNEL MACROS

A.2.1  dbg(mode, stuffToPassToPrinf)

dbg is the command to print out diagnostic messages. Note that debug modes are flags, thus you can OR them together.

\[
\text{dbg}(	ext{DBG_INIT | DBG_THR}, \text{("enqueueing 0x%p on runq\n", newthr)});
\]

A.2.2  Your DBG environment variable

The way you select which modes you are interested in is through the DBG environment variable. If you were interested in process and thread debugging messages you would say

\[
\text{setenv DBG proc,thr}
\]

A.2.3  Adding your own modes

To add your own debugging modes, you need to edit \texttt{dbg_modes.h}. You can define up to 128 modes. Simply add an entry at the top declaring your new debug mode, then in the second section, pair up your new mode with the text you will put in \$DBG to specify it.

A.3  Random kernel macros

Scattered around through various header files are some useful macros. Here’s a quick (and probably incomplete summary)

A.3.1  Debugging Macros

\textbf{KASSERT}(int)

Just like the old user-mode \texttt{assert} that you know and love, this fella when used frequently can be guaranteed to shave hours off of your debug time

\textbf{PANIC}(stuffToPassToPrintf)

When something is wrong, real wrong, and you can’t fix it, call this guy. It brings the system to a screeching halt and prints out your hopefully useful message.

\textbf{VERIFY}(int)

Similar to KASSERT. PANICs the kernel if what it is passed is \texttt{¡} 0. Useful to put around functions that can’t afford to fail.

A.3.2  VFS Macros

\textbf{VN_REF}(vnode_t*)

Increments the reference count on a vnode and adds an informative \texttt{dbg} with mode DBG_VNREF

\textbf{F_REF}(file_t*)

Same as VN_REF but for file_t*’s
A.3.3 Core Map Macros

CMAP_DIRTY_PAGE(pfn)
Looks up CMAP page from pfn and sets dirty

CMAP_CLEAN_PAGE(pfn)
Looks up CMAP page from pfn and clears dirty

CMAP_REF_PAGE(pfn)
Looks up CMAP page from pfn, increments the ref count and adds dbg with mode DBG_TLB

CMAP_RELEASE_PAGE(pfn)
Looks up CMAP page from pfn, clears dirty and calls page_free() on it

CMAP_DEREF_PAGE(pfn)
Looks up CMAP, decrements ref count and if == 0, calls core_release_page()

CMAP_BUMP_PAGE(pfn, val)
Looks up CMAP, increments ref count by val, and if == 0, calls core_release_page()

A.3.4 List Macros

These are described in detail in the header for list.h. Learn these well, as you will not get far without them. Also, save yourself some horrible syntax and use list_iterate_begin() and list_iterate_end() when possible.

A.3.5 Virtual Memory Macros

VFN_TO_VADDR(vfn), VADDR_TO_VFN(vaddr)
Converts between a frame number and an address (works for both virtual and physical)

VFN_TO_SEGNUM(vfn), SEGNUM_TO_VFN(segnum)
Converts between a segment number and a virtual frame number (you would only want to use this on a virtual frame number)

VFN_TO_PTOFFSET(vfn)
Returns the page table offset of a particular VFN

Other stuff
There are a host of additional macros and constants that you should check out in page.h
A.4 Kernel Datastructure Printing Functions

For each data structure in the kernel, there is a function which will print out its members in a (hopefully) meaningful manner. These functions all live in printdbg.h and printdbg.c. To use them, set your DBG mode to "print". They default to printing almost all information about a data structure. Feel free to eliminate parts of the printing if you find them too verbose.

A.4.1 Installing

Since this debugging facility was not ready at the beginning of the semester, you will have to retroactively add it to your project.

1. Copy printdbg.[ch] from /course/cs167/pub to the root directory of your project
2. Add printdbg.o to the OBJS line in your root Makefile
3. Edit the #define's at the top of printdbg.c to indicate which assignments you’ve completed
4. Remember, for any file you wish to call these functions from, you should 
   include “printdbg.h”

A.4.2 Using

To use, set your DBG mode to “print”. They default to printing almost all information about a data structure. Feel free to eliminate parts of the printing if you find them too verbose.

A.4.3 List of functions

- PRINT_LIST(list_t* list, jtype list, link_t* member, jvariable)
- PRINT_LIST_LINK(list_t* link)
- PRINT_KTQUEUE(ktqueue_t*)
- PRINT_BUF(buf_t*)
- PRINT_CORE(core_t*)
- PRINT_DEV(dev_t*)
- PRINT_KCOND(kcond_t*)
- PRINT_KMUTEX(kmutex_t*)
- PRINT_KTHREAD(kthread_t*)
- PRINT_PROC(proc_t*)
- PRINT_VMAREA(vm_area_t*)
- PRINT_PTE(pte_t)
- PRINT_PAGETABLE(pagetable_t*)
- PRINT_SEGTABLE(segtable_t*)
• PRINT_FILE(file_t*)
• PRINT_VFS(vfs_t*)
• PRINT_VNODE(vnode_t*)
• PRINT_TTY_RBUF(tty_rbuf_t*)
• PRINT_TTY_inode(tty_inode_t*)
• PRINT_S5_SUPER(s5_super_t*)
• PRINT_S5_inode(s5_inode_t*)
• PRINT_ADDRESS_SPACE(proc_t*)
Appendix A

Linux Kernel Coding Style

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won’t force my views on anybody, but this is what goes for anything that I have to be able to maintain, and I’d prefer it for most other things too. Please at least consider the points made here.

First off, I’d suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it’s a great symbolic gesture.

Anyway, here goes:

A.1 Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you’ve been looking at your screen for 20 straight hours, you’ll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you’re screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you’re nesting your functions too deep. Heed that warning.

A.2 Placing Braces

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the

---

1This was L\TeX\'ed up by me, Keith Adams. Linus himself may or may not approve. Linus gets pretty inflammatory in places, but he also makes some good points. Take this with a grain of salt or two.
preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```c
if (x is true) {
    we do y
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```c
int function(int x)
{
    body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are right and (b) K&R are right. Besides, functions are special anyway (you can’t nest them in C).

Note that the closing brace is empty on a line of its own, except in the cases where it is followed by a continuation of the same statement, ie a ”while” in a do-statement or an ”else” in an if-statement, like this:

```c
do {
    body of do-loop
} while (condition);
```

and

```c
if (x == y) {
    ..
} else if (x > y) {
    ....
} else {
    ....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

### A.3 Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like ThisVariableIsATemporaryCounter. A C programmer would call that variable ”tmp”, which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function ”foo” is a shooting offense.
GLOBAL variables (to be used only if you really need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that "count_active_users()" or similar, you should not call it "cntusr()".

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called "i". Calling it "loop_counter" is non-productive, if there is no chance of it being mis-understood. Similarly, "tmp" can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See next section.

A.4 Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's ok to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it’s performance-critical, and it will probably do a better job of it that you would have done).

Another measure of the function is the number of local variables. They shouldn’t exceed 5-10, or you’re doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you’re brilliant, but maybe you’d like to understand what you did 2 weeks from now.

A.5 Commenting

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it’s much better to write the code so that the working is obvious, and it’s a waste of time to explain badly written code.

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to section A.4 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.
A.6 You’ve made a mess of it

That’s ok, we all do. You’ve probably been told by your long-time unix user helper that "GNU emacs" automatically formats the C sources for you, and you’ve noticed that yes, it does do that, but the defaults it uses are less than desirable (in fact, they are worse than random typing - a infinite number of monkeys typing into GNU emacs would never make a good program).

So, you can either get rid of GNU emacs, or change it to use saner values. I did the first, so don’t ask me how to do the latter. But even if you fail in getting emacs to do sane formatting, not everything is lost: use "indent".

Now, again, GNU indent has the same brain dead settings that GNU emacs has, which is why you need to give it a few command line options. However, that’s not too bad, because even the makers of GNU indent recognize the authority of K&R (the GNU people aren’t evil, they are just severely misguided in this matter), so you just give indent the options "-kr -i8" (stands for "K&R, 8 character indents").

"indent" has a lot of options, and especially when it comes to comment re-formatting you may want to take a look at the manual page. But remember: "indent" is not a fix for bad programming.
Index

chdir(2), 12
create(), 14
exit(), 7
file_t, 12, 14, 15
file_t’s, 15
file_t’s, 15
fork(), 7
getpage(), 14
kt_quantum, 8, 9
kthread.h, 8
kthread_t, 8
link/unlink(), 14
lookup(), 12, 14
main(), 7
mkdir/rmdir(), 14
p_files, 8
p_nchld, 8
p_pid, 7
p_pproc, 7
p_state, 8, 8
p_status, 7
p_threads, 7
proc.c, 7
proc.h, 7, 8
PROC_DEAD, 8
PROC_NONE, 8
PROC_RUNNING, 8
proc_t, 7, 15
read(), 12
swtch(), 8
vfs_t, 13, 13
vfs_t, 12
vfs_type_t, 13
vfs_type_t, 12
vnode_t, 12, 13, 13, 14, 15
vnode_t’s, 15
vnode_t, 14
vnode_t, 13
vnode_t, 14
vput/fput(), 15
wait(), 8
write(), 12