Adopting and Commenting the Old Kernel Source Code for Education

Jiong Zhao
University of TongJi, Shanghai
gohigh@gmail.com

Trent Jarvi
University of Denver
taj@www.linux.uk.org

Abstract

Dissecting older kernels including their problems can be educational and an entertaining review of where we have been. In this session, we examine the older Linux kernel version 0.11 and discuss some of our findings. The primary reason for selecting this historical kernel is that we have found that the current kernel's vast quantity of source code is far too complex for hands-on learning purposes. Since the 0.11 kernel has only 14,000 lines of code, we can easily describe it in detail and perform some meaningful experiments with a runnable system effienctly. We then examine several aspects of the kernel including the memory management, stack usage and other aspects of the Linux kernel. Next we explain several aspects of using Bochs emulator to perform experiments with the Linux 0.11 kernel. Finally, we present and describe the structure of the Linux kernel source including the `lib/` directory.

1   Introduction

As Linus once said, if one wants to understand the details of a software project, one should "RTFSC—Read The F**king Source Code." The kernel is a complete system, the parts relate to each other to fulfill the functions of a operating system. There are many hidden details in the system. If one ignores these details, like a blind men trying to size up the elephant by taking a part for the whole, its hard to understand the entire system and is difficult to understand the design and implementations of an actual system. Although one may obtain some of the operating theory through reading classical books like the "The design of Unix operating system," [4] the composition and internal relationships in an operating system are not easy to comprehend. Andrew Tanenbaum, the author of MINIX[1], once said in his book, "teaching only theory leaves the student with a lopsided view of what an operating system is really like." and "Subjects that really are important, such as I/O and file systems, are generally neglected because there is little theory about them." As a result, one may not know the tricks involved in implementing a real operating system. Only after reading the entire source code of a operating system, may one get a feeling of sudden enlightened about the kernel.

In 1991 Linus made a similar statements[5] after distributing kernel version 0.03, "The *GNU* kernel (Hurd) will be free, but is currently not ready, and will be too big to understand and learn." Likewise, the current Linux kernel is too large to easily understand. Due to the small amount of code (only 14,000 lines) as shown in Figure 1, the usability and the consistency

with the current kernel, it is feasible to choose Linux 0.11 kernel for students to learn and perform experiments. The features of the 0.11 kernel are so limited, it doesn't even contain job control or virtual memory swapping. It can, however, still be run as a complete operating system. As with an introductory book on operating systems, we need not deal with the more complicated components such as *VFS, ext3*, networking and more comprehensive memory management systems in a modern kernel. As students understand of the main concepts concerning how an operating system is generally implemented, they can learn to understand the advanced parts for themselves. Thus, both the teaching and learning become more efficient and consume considerably less time. The lower barrier to entry for learning can even stimulate many young people to take part in and involve in the Linux activities.
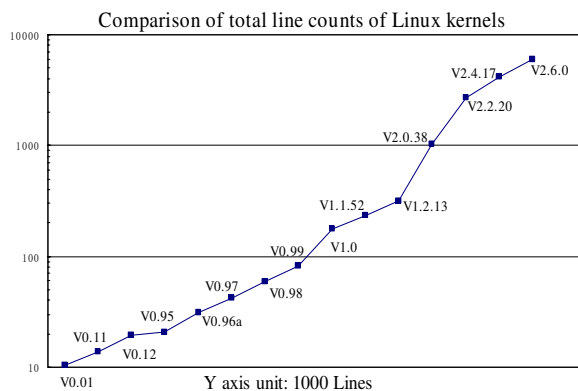

Figure 1: Lines of code in various kernel versions

From teaching experience and student feedback, we found the most difficult part of studying the 0.11 kernel is the memory management. Therefore, in the following sections we mainly deal with how the 0.11 kernel manages and uses memory in the protected mode of the Intel *IA-32* processor along with the different kinds of stacks used during the kernel initialization of each task.

2   Linux Kernel Architecture

The Linux kernel is composed of five modules: task scheduling, memory management, file system, interprocess communication (IPC) and network. The task scheduling module is responsible for controlling the usage of the processor for all tasks in the system. The strategy used for scheduling is to provide reasonable and fair usage between all tasks in the system while at the same time insuring the processing of hardware operations. The memory management module is used to insure that all tasks can share the main memory on the machine and provide the support for virtual memory mechanisms. The file system module is used to support the driving of and storage in peripheral devices. Virtual file system modules hide the various differences in details of the hardware devices by providing a universal file interface for peripheral storage equipment and providing support for multiple formats of file systems. The *IPC* module is used to provide the means for exchanging messages between processes. The network interface module provides access to multiple communication standards and supports various types of network hardware.

The relationship between these modules is illustrated in Figure 2. The lines between them indicates the dependences of each other. The dashed lines and dashed line box indicate the part not implemented in Linux 0.1x.

The figure shows the scheduling module relationship with all the other modules in the kernel since they all depend on the schedules provided to suspend and restart their tasks. Generally, a module may hang when waiting for hardware operations, and continue running after the hardware operation finishes. The other three modules have like relationships with the schedule module for similar reasons.
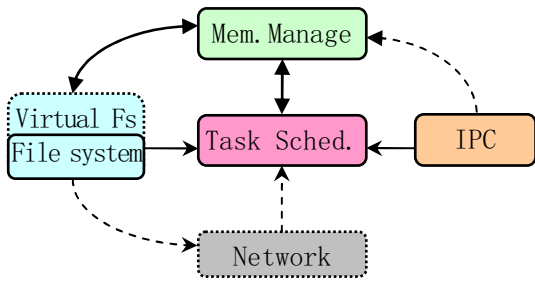
Figure 2: The relationship between Linux kernel modules

The remaining modules have implicit dependences with each other. The scheduling subsystem needs memory management to adjust the physical memory space used by each task. The IPC subsystem requires the memory management module to support shared memory communication mechanisms. Virtual file systems can also use the network interface to support the network file system (*NFS*). The memory management subsystem may also use the file system to support the swapping of memory data blocks.

From the monolithic model structure, we can illustrate the main kernel modules in Figure 3 based on the structure of the Linux 0.11 kernel source code.

3 Memory Usage

In this section, we first describe the usage of physical memory in Linux 0.1x kernel. Then we explain the memory *segmentation, paging, multitasking* and the *protection* mechanisms. Finally, we summarize the relationship between virtual, linear, and physical address for the code and data in the kernel and for each task.
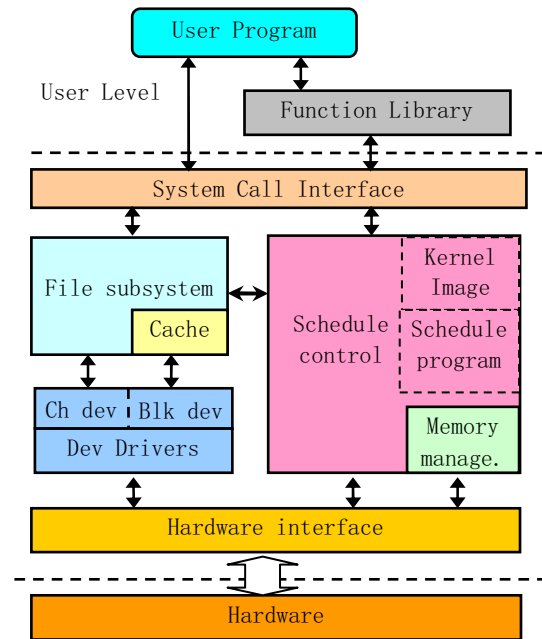

Figure 3: Kernel structure framework

3.1 Physical Memory

In order to use the physical memory of the machine efficiently with Linux 0.1x kernel, the memory is divided into several areas as shown in Figure 4.
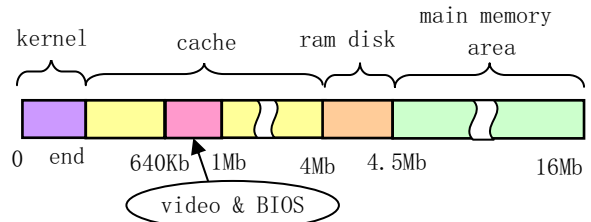

Figure 4: The regions of physical memory

As shown in Figure 4, the kernel code and data occupies the first portion of the physical memory. This is followed by the cache used for block devices such as hard disks and floppy drives eliminating the memory space used by the adapters and *ROM BIOS*. When a

task needs data from a block device, it will be first read into the cache area from the block device. When a task needs to output the data to a block device, the data is put into the cache area first and then is written into the block device by the hardware driver in due time. The last part of the physical memory is the main area used dynamically from programs. When kernel code needs a free memory page, it also needs to make a request from the memory management subsystem. For a system configured with virtual *RAM disks* in physical memory, space must be reserved in memory.

Physical memory is normally managed by the processor's memory management mechanisms to provide an efficient means for using the system resources. The Intel 80X86 CPU provides two memory management mechanisms: Segmentation and paging. The paging mechanism is optional and its use is determined by the system programmer. The Linux operating system uses both memory segmentation and paging mechanism approaches for flexibility and efficiency of memory usage.

3.2   Memory address space

To perform address mapping in the Linux kernel, we must first explain the three different address concepts used in *virtual* or *logical* address space, the CPU *linear* address space, and the actual *physical* address space. The *virtual* addresses used in virtual address space are addresses composed of the *segment selector* and *offset* in the segment generated by program. Since the two part address can not be used to access physical memory directly, this address is referred to as a virtual address and must use at least one of the address translation mechanisms provided by CPU to map into the physical memory space. The virtual address space is composed of the *global address space* addressed by the descriptors in global descriptor

table (*GDT*) and the *local address space* addressed by the local descriptor table (*LDT*). The index part of a *segment selector* has thirteen bits and one bit for the table index. The Intel 80X86 processor can then provide a total of 16384 selectors so it can addresses a maximum of 64T of virtual address space[2]. The logical address is the offset portion of a virtual address. Sometimes this is also referred to as virtual address.

Linear address is the middle portion of address translation from virtual to physical addresses. This address space is addressable by the processor. A program can use a *logical address* or *offset* in a segment and the base address of the segment to get a linear address. If *paging* is enabled, the linear address can be translated to produced a physical address. If the *paging* is disabled, then the linear address is actually the same as physical address. The linear address space provided by Intel 80386 is 4 GB.

Physical address is the address on the processor's external address bus, and is the final result of address translation.

The other concept that we examine is *virtual memory*. Virtual memory allows the computer to appear to have more memory than it actually has. This permits programmers to write a program larger than the physical memory that the system has and allows large projects to be implemented on a computer with limited resources.

3.3   Segmentation and paging mechanisms

In a segmented memory system, the logical address of a program is automatically mapped or translated into the middle 4 GB linear address space. Each memory reference refers to the memory in a segment. When programs reference a memory address, a linear address is produced by adding the segment base address with
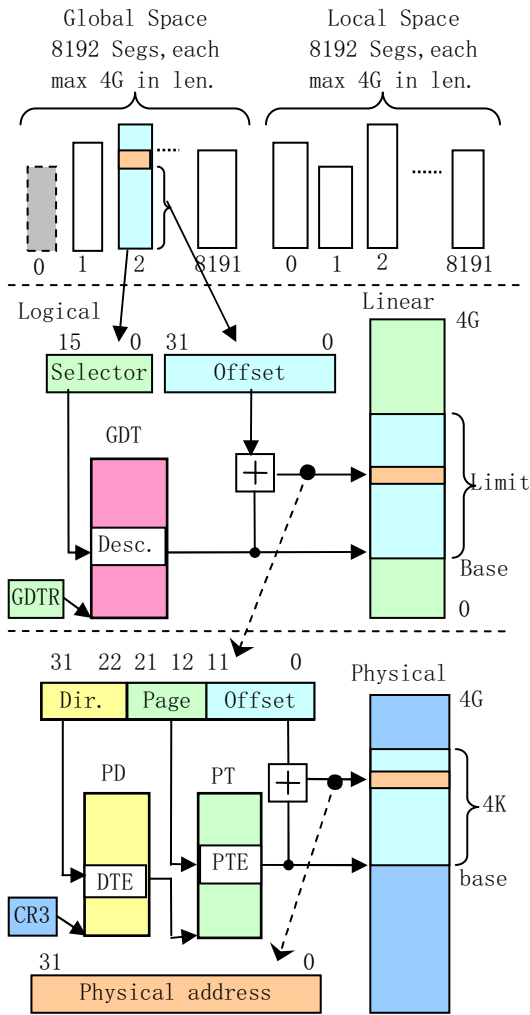
Figure 5: The translation between virtual or logical, linear and physical address

pages of 4 KB. When programs request memory, the processor allocates memory in pages for the program.

Since Linux 0.1x kernel uses only one *page directory*, the mapping function from linear to physical space is same for the kernel and processes. To prevent tasks from interfering with each other and the kernel, they have to occupy different ranges in the linear address space. The Linux 0.1x kernel allocates 64MB of linear space for each task in the system, the system can therefor hold at most 64 simultaneous tasks (64MB * 64 = 4G) before occupying the entire Linear address space as illustrated in Figure 6.

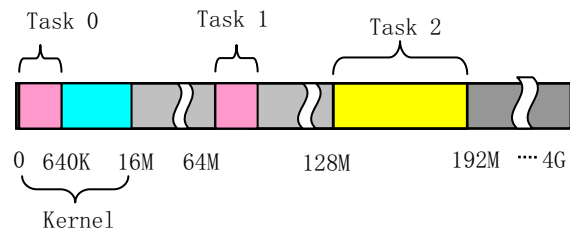Figure 6: The usage of linear address space in the Linux 0.1x kernel

the logical address visible to the programmer. If *paging* is not enabled, at this time, the linear address is sent to the external address bus of the processor to access the corresponding physical address directly.

If *paging* is enabled on the processor, the linear address will be translated by the *paging* mechanism to get the final physical corresponding physical address. Similar to the segmentation, paging allow us to relocate each memory reference. The basic theory of paging is that the processor divides the whole linear space into

3.4 The relationship between virtual, linear and physical address

We have briefly described the memory segmentation and paging mechanisms. Now we will examine the relationship between the kernel and tasks in virtual, linear and physical address space. Since the creation of *tasks 0* and *1* are special, we'll explain them separately.

3.4.1 The address range of kernel

For the code and data in the Linux 0.1x kernel, the initialization in `head.s` has already set the limit for the kernel and data segments to

be 16MB in size. These two segments overlap at the same linear address space starting from address 0. The *page directory* and *page table* for kernel space are mapped to 0-16MB in physical memory (the same address range in both spaces). This is all of the memory that the system contains. Since one page table can manage or map 4MB, the kernel code and data occupies four entries in the *page directory*. In other words, there are four secondary page tables with 4MB each. As a result, the address in the kernel segment is the same in the physical memory. The relationship of these three address spaces in the kernel is depicted in Figure 7.
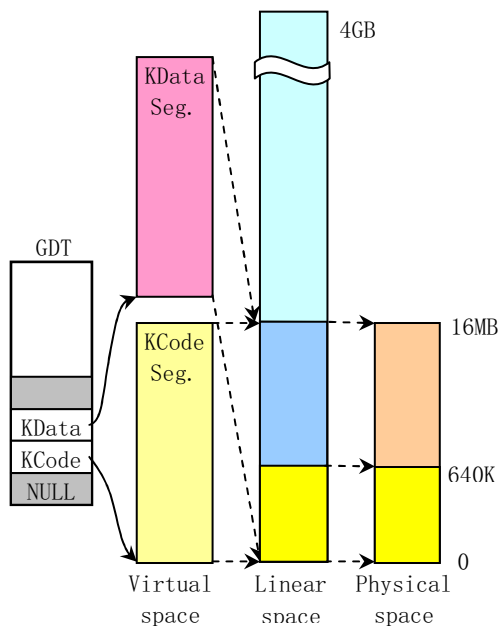

Figure 7: The relationship of the three address spaces in a 0.1x kernel

As seen in Figure 7, the Linux 0.1x kernel can manage at most 16MB of physical memory in 4096 page frames. As explained earlier, we know that: (1) the address range of kernel code and data segments are the same as in the physical memory space. This configuration can greatly reduce the initialization operations the kernel must perform. (2) *GDT* and Interrupt Descriptor Table (*IDT*) are in the kernel data segment, thus they are located in the same address in both address spaces. In the execution of code in `setup.s` in real mode, we have setup both temporary *GDT* and *IDT* at once. These are required before entering protected mode. Since they are located by physical address `0x90200` and this will be overlapped and used for block device cache, we have to recreate *GDT* and *IDT* in `head.s` after entering protected mode. The segment selectors need to be reloaded too. Since the locations of the two tables do not change after entering protected mode, we do not need to move or recreate them again. (3) All tasks except *task 0* need additional physical memory pages in different linear address space locations. They need the memory management module to dynamically setup their own mapping entries in the *page directory* and *page table*. Although the code and static data of *task 1* are located in kernel space, we need to obtain new pages to prevent interference with *task 0*. As a result, *task 1* also needs its own page entries.

While the default manageable physical memory is 16MB, a system need not contain 16MB memory. A machine with only 4MB or even 2MB could run Linux 0.1x smoothly. For a machine with only 4MB, the linear address range 4MB to 16MB will be mapped to nonexistent physical space by the kernel. This does not disrupt or crash the kernel. Since the kernel knows the exact physical memory size from the initialization stage, no pages will be mapped into this nonexistent physical space. In addition, since the kernel has limited the maximum physical memory to be 16MB at boot time (in `main()` corresponding to `startkernel()`), memory over 16MB will be left unused. By adding some page entries for the kernel and changing some of the kernel source, we certainly can make Linux 0.1x support more physical memory.

3.4.2 The address space relationship for task 0

Task 0 is artificially created or configured and run by using a special method. The limits of its code and data segments are set to the 640KB included in the kernel address space. Now *task 0* can use the kernel page entries directly without the need for creating new entries for itself. As a result, its segments are overlapped in linear address space too. The three space relationship is shown in Figure 8.
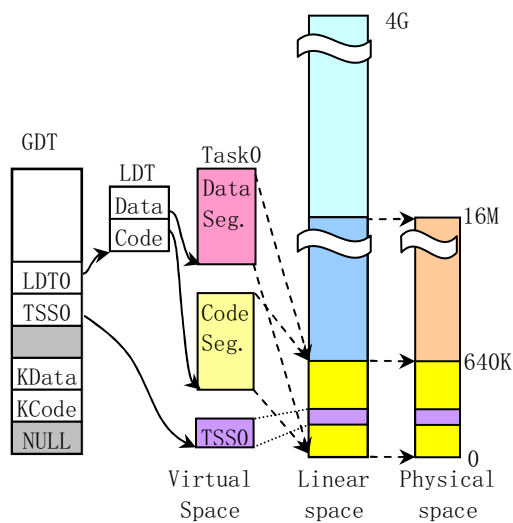

Figure 8: The relationship of three address spaces for task 0

As *task 0* is totally contained in the kernel space, there is no need to allocate pages from the main memory area for it. The kernel stack and the user stack for *task 0* are included the kernel space. *Task 0* still has read and write rights in the stacks since the page entries used by the kernel space have been initialized to be readable and writable with user privileges. In other words, the flags in page entries are set as *U/S=1, R/W=1*.

3.4.3 The address space relationship for task 1

Similar to *task 0*, *task 1* is also a special case in which the code and data segment are included in kernel module. The main difference is that when forking *task 1*, one free page is allocated from the main memory area to duplicate and store *task 0*'s page table entries for *task 1*. As a result, *task 1* has its own *page table* entries in the *page directory* and is located at range from 64MB to 128MB (actually 64MB to 64MB + 640KB) in linear address space. One additional page is allocated for task 1 to store its *task structure (PCB)* and is used as its kernel mode stack. The task's *Task State Segment (TSS)* is also contained in task's structure as illustrated in Figure 9.
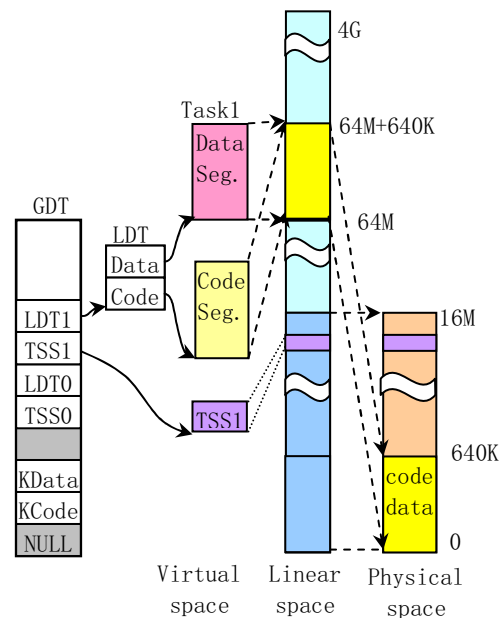

Figure 9: The relationship of the three address spaces in task 1

Task 1 and *task 0* will share their user stack `user_stack[]` (refer to `kernel/sched. c`, lines 67-72). Thus, the stack space should be "clean" before *task 1* uses it to ensure that there

is no unnecessary data on the stack. When fork-ing *task 1*, the user stack is shared between *task 0* and *task 1*. However when *task 1* starts run-ning, the stack operating in *task 1* would cause the processor to produce a page fault because the page entries have been modified to be read only. The memory management module will therefor need allocate a free page for *task 1's* stack.

3.4.4 The address space relationship for other tasks

For *task 2* and higher, the parent is *task 1* or the *init* process. As described earlier, Linux 0.1x can have 64 tasks running synchronously in the system. Now we will detail the address space usage for these additional tasks.

Beginning with *task 2*, if we designate nr as the task number, the starting location for *task* nr will be at nr * 64MB in linear address space. *Task 2*, for example, begins at address 2 * 64MB = 128MB in the linear address space, and the limits of code and data segments are set to 64MB. As a result, the address range occu-pied by *task 2* is from 128MB to 192MB, and has 64MB/4MB = 16 entries in the page direc-tory. The code and data segments both map to the same range in the linear address space. Thus they also overlap with the same address range as illustrated in Figure 10.

After *task 2* has forked, it will call the func-tion execve() to run a shell program such as bash. Just after the creation of *task 2* and be-fore call execve(), *task 2* is similar to *task 1* in the three address space relationship for code and data segments except the address range oc-cupied in linear address space has the range from 128MB to 192MB. When *task 2's* code calls execve() to load and run a shell pro-gram, the page entries are copied from *task 1* and corresponding memory pages are freed and
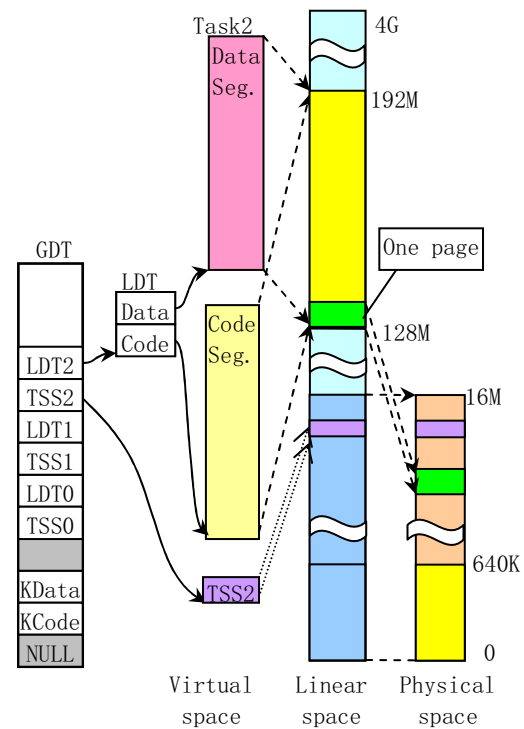

Figure 10: The relationship of the three address spaces in tasks beginning with task 2

new page entries are set for the shell program. Figure 10 shows this address space relation-ship. The code and data segment for *task 1* are replaced with that of the shell program, and one physical memory page is allocated for the code of the shell program. Notice that although the kernel has allocated 64MB linear space for *task 2*, the operation of allocating actual physical memory pages for code and data segments of the shell program is delayed until the program is running. This delayed allocation is called de-mand paging.

Beginning with kernel version 0.99.x, the usage of memory address space changed. Each task can use the entire 4G linear space by changing the page directory for each tasks as illustrated in Figure 11. There are even more changes are in current kernels.
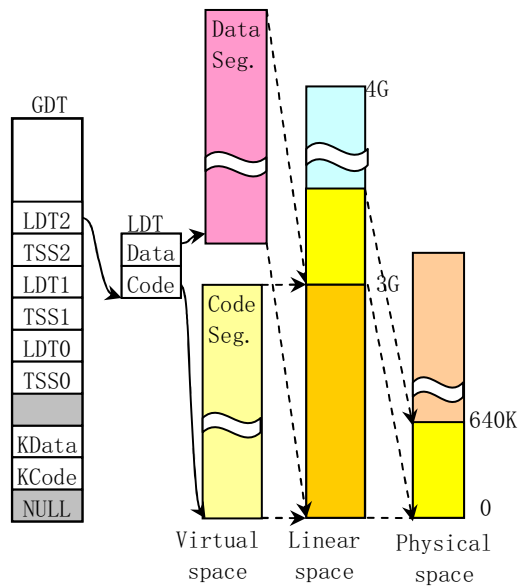
Figure 11: The relationship of the three address space for tasks in newer kernels

4   Stack Usage

This section describes several different methods used during the processing of kernel booting and during normal task stack operations. Linux 0.1x kernel uses four different kinds of stacks: the temporary stack used for system booting and initialization under real address mode; The kernel initialization stack used after the kernel enters protected mode, and the user stack for *task 0* after moving into task 0; The kernel stack of each task used when running in the kernel and the user stacks for each task except for *tasks 0 and 1*.

There are two main reasons for using four different stacks (two used only temporarily for booting) in Linux. First, when entering protected from real mode, the addressing method used by the processor has changed. Thus the kernel needs to rearrange the stack area. In addition, to solve the protection problems brought by the new privilege level on processor, we need to use different stacks for kernel code at

level 0 and for user code at level 3 respectively. When a task runs in the kernel, it uses the kernel mode stack pointed by the values in `ss0` and `esp0` fields of its *TSS* and stores the task's user stack pointer in this stack. When the control returns to the user code or to level 3, the user stack pointer will be popped out, and the task continues to use the user stack.

4.1   Initialization period

When the *ROM BIOS* code boots and loads the bootsect into memory at physical address `0x7C00`, no stack is used until it is moved to the location `0x9000:0`. The stack is then set at `0x9000:0xff00`. (refer to line 61–62 in `boot/bootsect.s`). After control is transferred to `setup.s`, the stack remains unchanged.

When control is transferred to `head.s`, the processor runs in protected mode. At this time, the stack is setup at the location of `user_stack[]` in the kernel code segment (line 31 in `head.s`). The kernel reserves one 4 KB page for the stack defined at line 67–72 in `sched.c` as illustrated in Figure 12.

This stack area is still used after the control transfers into `init/main.c` until the execution of `move_to_user_mode()` to hand the control over to *task 0*. The above stack is then used as a user stack for *task 0*.

4.2   Task stacks

For the processor privilege levels 0 and 3 used in Linux, each task has two stacks: kernel mode stack and user mode stack used to run kernel code and user code respectively. Other than the privilege levels, the main difference is that the size of kernel mode stack is smaller than that of the user mode stack. The former is located
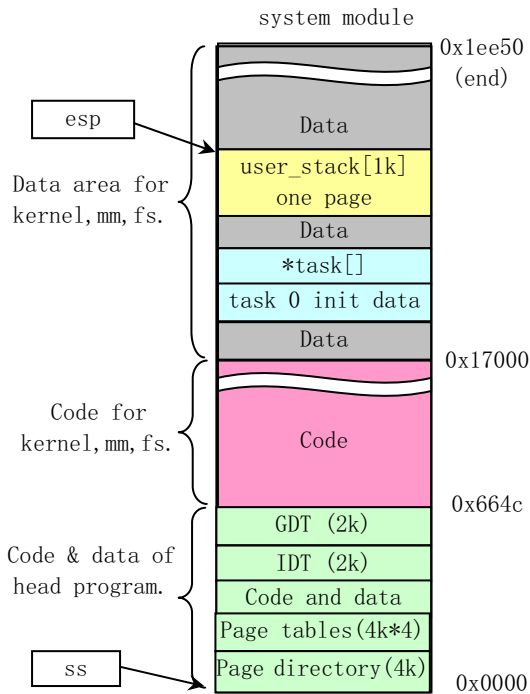
system module

```
                          0x1ee50
                           (end)
            Data
esp ──→
        user_stack[1k]
          one page
            Data
          *task[]
       task 0 init data
            Data
                          0x17000

            Code

                          0x664c
         GDT  (2k)
         IDT  (2k)
       Code and data
      Page tables(4k*4)
      Page directory(4k)
ss ──→                    0x0000
```

Data area for kernel, mm, fs.

Code for kernel, mm, fs.

Code & data of head program.

Figure 12: The stack used for kernel code after entering protected mode

```
 Code                    Cmd Line  Environ.
and data      ))         Params.   Params.
0                    ↑      ↑              64M
              Current    Stack
                esp      bottom
```

Figure 13: User stack in task's logical space

stack. Then the memory manager will allocate and duplicate the stack page for the task.

Similar to the user stack, each task has its own *kernel mode stack* used when operating in the kernel code. This stack is located in the memory to pointed by the values in ss0, esp0 fields in task's *TSS*. ss0 is the stack segment selector like the *data selector* in the kernel. esp0 indicates the stack bottom. Whenever control transfers to the kernel code from user code, the kernel mode stack for the task always starts from ss0:esp0, giving the kernel code an empty stack space. The bottom of a task's kernel stack is located at the end of a memory page where the task's data structure begins. This arrangement is setup by making the privilege level 0 stack pointer in *TSS* point to the end of the page occupied by the task's data structure when forking a new task. Refer to line 93 in kernel/fork.c as below:

```
p->tss.esp0 = PAGE_SIZE+(long)p;
p->tss.ss0 = 0x10;
```

p is the pointer of the new task structure, tss is the structure of the task status segment. The kernel request a free page to store the task structure pointed by p. The tss structure is a field in the task structure. The value of tss.ss0 is set to the selector of kernel data segment and the tss.esp0 is set to point to the end of the page as illustrated in Figure 14.

As a matter of fact, tss.esp0 points to the byte outside of the page as depicted in the fig-

at the bottom in a page coexisting with task's structure, and no more than 4KB in size. The later can grow down to nearly 64MB in user space.

As described, each task has its own 64MB logical or linear address space except for *task 0* and *1*. When a task was created, the bottom of its user stack is located close to the end of the 64MB space. The top portion of the user space contains additional environmental parameters and command line parameters in a backwards orientation, and then the user stack as illustrated in Figure 13.

Task code at privilege level 3 uses this stack all of the time. Its corresponding physical memory page is mapped by paging management code in the kernel. Since Linux utilizes the *copy-on-write*[3] method, both the parent and child process share the same user stack memory until one of them perform a write operation on the
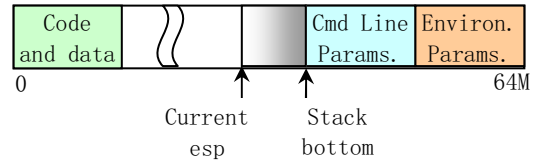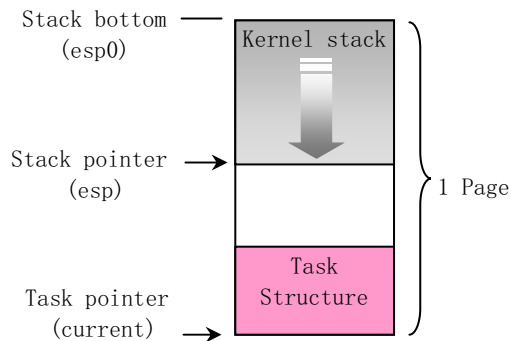
Figure 14: The kernel mode stack of a task

ure. This is because the Intel processor decreases the pointer before storing a value on the stack.

4.3 The stacks used by task 0 and task 1

Both *task 0* or *idle* task and *task 1* or *init* task have some special properties. Although *task 0* and *task 1* have the same code and data segment and 640KB limits, they are mapped into different ranges in linear address space. The code and data segments of *task 0* begins at address 0, and *task 1* begins at address 64MB in the linear space. They are both mapped into the same physical address range from 0 to 640KB in kernel space. After calling the function `move_to_user_mode()`, the kernel mode stacks of *task 0* and *task 1* are located at the end of the page used for storing their task structures. The user stack of *task 0* is the same stack originally used after entering protected mode; the space for `user_stack[]` array defined in `sched.c` program. Since *task 1* copies *task 0*'s user stack when forking, they share the same stack space in physical memory. When *task 1* starts running, however, a page fault exception will occur when *task 1* writes to its user stack because the page entries for *task 1* have been initialized as read-only. At this moment, the kernel will allocate a free page

in main memory area for the stack of *task 1* in the exception handler, and map it to the location of *task 1*'s user stack in the linear space. From now on, *task 1* has its own separate user stack page. As a result, the user stack for *task 0* should be "clean" before *task 1* uses the user stack to ensure that the page of stack duplication does not contain useless data for *task 1*.

The kernel mode stack for *task 0* is initialized in its static data structure. Then its user stack is set up by manipulating the contents of the stack originally used after entering protected mode and emulating the interrupt return operation using `IRET` instruction as illustrated in Figure 15.
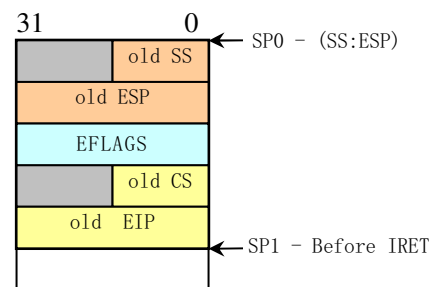

Figure 15: Stack contents while returning from privilege level 0 to 3

As we know, changing the privilege level will change the stack and the old stack pointers will be stored onto the new stack. To emulate this case, we first push the *task 0*'s stack pointer onto the stack, then the pointer of the next instruction in *task 0*. Finally we run the `IRET` instruction. This causes the privilege level change and control to be transferred to *task 0*. In the Figure 15, the old *SS* field stores the data selector of *LDT* for *task 0* (0x17) and the old *ESP* field value is not changed since the stack will be used as the user stack for *task 0*. The old *CS* field stores the code selector (0x0f) for *task 0*. The old *EIP* points to the next instruction to be executed. After the manipulation, a *IRET* instruction switches the privileges

from level 0 to level 3. The kernel begins running in *task 0*.

4.4 Switch between kernel mode stack and user mode stack for tasks

In the Linux 0.1x kernel, all interrupts and exceptions handlers are in mode 0 so they belong to the operating system. If an interrupt or exception occurs while the system is running in user mode, then the interrupt or exception will cause a privilege level change from level 3 to level 0. The stack is then switched from the user mode stack to the kernel mode stack of the task. The processor will obtain the kernel stack pointers `ss0` and `esp0` from the task's *TSS* and store the current user stack pointers into the task's kernel stack. After that, the processor pushes the contents of the current *EFLAGS* register and the next instruction pointers onto the stack. Finally, it runs the interrupt or exception handler.

The kernel *system call* is trapped by using a software interrupt. Thus an `INT 0x80` will cause control to be transferred to the kernel code. Now the kernel code uses the current task's kernel mode stack. Since the privilege level has been changed from level 3 to level 0, the user stack pointer is pushed onto the kernel mode stack, as illustrated in Figure 16.

If a task is running in the kernel code, then an interrupt or exception never causes a stack switch operation. Since we are already in the kernel, an interrupt or exception will never cause a privilege level change. We are using the kernel mode stack of the current task. As a result, the processor simply pushes the *EFLAGS* and the return pointer onto the stack and starts running the interrupt or exception handler.
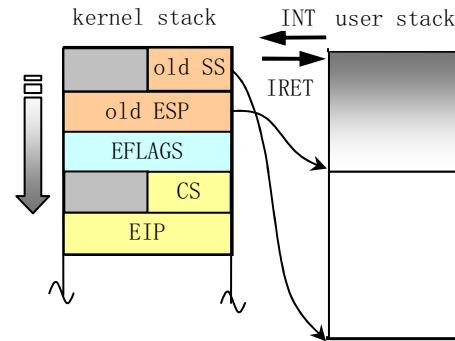

Figure 16: Switching between the kernel stack and user stack for a task

5 Kernel Source Tree

Linux 0.11 kernel is simplistic so the source tree can be listed and described clearly. Since the 0.11 kernel source tree only has 14 directories and 102 source files it is easy to find specific files in comparison to searching the much larger current kernel trees. The main `linux/` directory contains only one Makefile for building. From the contents of the Makefile we can see how the kernel image file is built as illustrated in Figure 17.
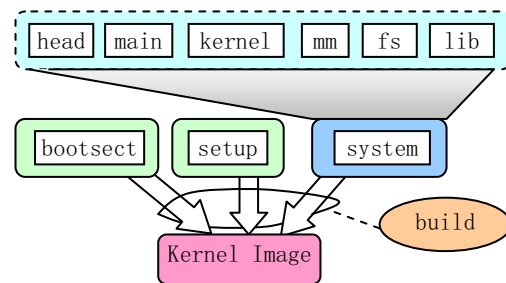

Figure 17: Kernel layout and building

There are three assembly files in the `boot/` directory: `bootsect.s`, `setup.s`, and `head.s`. These three files had corresponding files in the more recent kernel source trees until 2.6.x kernel. The `fs/` directory contains source files for implementing a *MINIX* version

1.0 file system. This file system is a clone of the traditional UN*X file system and is suitable for someone learning to understand how to implement a usable file system. Figure 18 depicts the relationship of each files in the `fs/` directory.
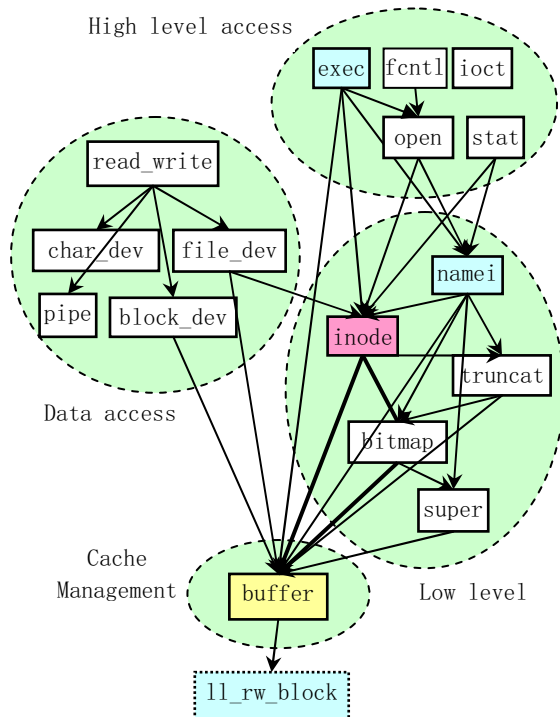

Figure 18: File relationships in fs/ directory

The `fs/` files can be divided into four types. The first is the block cache manager file `buffer.c`. The second is the files concerning with low level data operation files such `inode.c`. The third is files used to process data related to char, block devices and regular files. The fourth is files used to execute programs or files that are interfaces to user programs.

The `kernel/` directory contains three kinds of files as depicted in Figure 19.

The first type is files which deal with hardware interrupts and processor exceptions. The second type is files manipulating system calls from
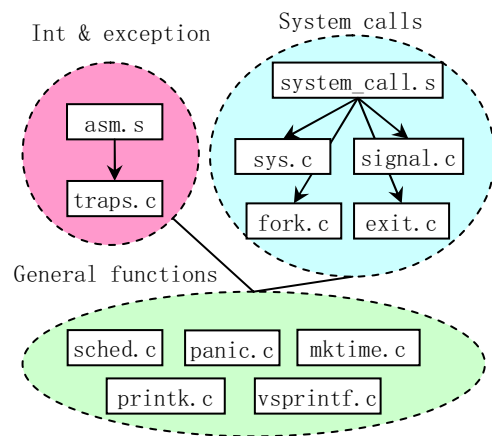

Figure 19: Files in the kernel/ directory

user programs. The third category is files implementing general functions such as scheduling and printing messages from the kernel.

Block device drivers for hard disks, floppy disks and ram disks reside in a subdirectory `blk_drv/` in the `kernel/`, thus the Linux 0.11 kernel supports only three classical block devices. Because Linux evolved from a terminal emulation program, the serial terminal driver is also included in this early kernel in addition to the necessary console character device. Thus, the 0.11 kernel contains at least two types of char device drivers as illustrated in Figure 20.

The remaining directories in the kernel source tree include, `init`, `mm`, `tools`, and `math`. The `include/` contains the head files used by the other kernel source files. `init/` contains only the kernel startup file `main.c`, in which, all kernel modules are initialized and the operating system is prepared for use. The `mm/` directory contains two memory management files. They are used to allocate and free pages for the kernel and user programs. As mentioned, the mm in 0.11 kernel uses demand paging technology. The `math/` directory only contains math source code stubs as 387 emula-
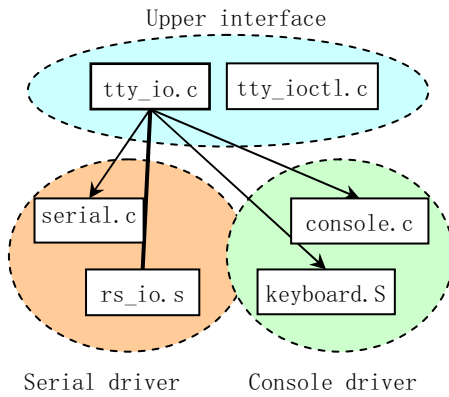
Figure 20: Character devices in Linux 0.11 kernel

tion did not appear until the 0.12 kernel.

6 Experiments with the 0.1x kernel

To facilitate understanding of the Linux 0.11 kernel implementation, we have rebuilt a runnable Linux 0.11 system, and designed several experiments to watch the kernel internal activities using the *Bochs PC emulator*. *Bochs* is excellent for debugging operating systems. The *Bochs* software package contains an internal debugging tool, which we can use to observe the dynamic data structures in the kernel and examine the contents of each register on the processor.

It is an interesting exercise to install the Linux 0.11 system from scratch. It is a good learning experience to build a root file system image file under Bochs.

Modifying and compiling the kernel source code are certainly the most important experiments for learning about operating systems. To facilitate the process, we provide two environments in which, one can easily compile the kernel. One is the original *GNU* gcc environment

under Linux 0.11 system in Bochs. The other is for more recent Linux systems such as *Red Hat 9* or *Fedora*. In the former environment, the 0.11 kernel source code needs no modifications to successfully compile. For the later environment one needs to modify a few lines of code to correct syntax errors. For people familiar with *MASM* and *VC* environment under windows, we even provide modified 0.11 kernel source code that can compile. Offering source code compatible with multiple environments and providing forums for discussion helps popularize linux and the linux community with new people interested in learning about operating systems :-)

7 Summary

From observing people taking operating system courses with the old Linux kernel, we found that almost all the students were highly interested in the course. Some of them even started programming their own operating systems.

The 0.11 kernel contains only the basic features that an operating system must have. As a result, there are many important features not implemented in 0.11 kernel. We now plan to adopt either the 0.12 or 0.98 kernel for teaching purposes to include job control, virtual *FS*, virtual console and even network functions. Due to time limitations in the course, several simplifications and careful selection of material will be needed.

References

[1] Albert S. Woodhull Andrew S. Tanenbaum. *OPERATING SYSTEMS: Design and Implementation*. Prentice-Hall, Inc., 1997.

[2] Patrick P. Gelsinger John H. Crawford.
Programming the 80386. SYBEX Inc.,
1987.

[3] Robert Love. *Linux Kernel Development*.
Sams Inc., 2004.

[4] M.J.Bach. *The Design of Unix Operating
System*. Prentice-Hall, Inc., 1986.

[5] Linus Torvalds. LINUX – a free unix-386
kernel. October 1991.

Proceedings of the
Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*