Notes on ECE 391, UIUC

Operating System and Computer Networking

Author
Hao Bai, UIUC

Professors
Prof. Zbigniew T Kalbarczyk, ECE391
Prof. Yih-Chun Hu, ECE391

October 10, 2021

Contents

Notation

This section provides a concise reference describing notation used throughout this document. For texts, we use **bold** text to indicate a new terminology, *italic* to indicate quotes (e.g., *dummy face* means "dummy face"), and `consolas` to indicate programming code.

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
A	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor A
$\mathsf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$\mathrm{a} \perp \mathrm{b}$	The random variables a and b are independent
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	They are conditionally independent given c
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E} f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \| Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\zeta(x)$	Softplus, $\log(1+\exp(x))$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$
$\mathbf{1}_{\text{condition}}$	is 1 if the condition is true, 0 otherwise

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathsf{X})$. This denotes the application of f to the array element-wise. For example, if $\mathsf{C} = \sigma(\mathsf{X})$, then $\mathsf{C}_{i,j,k} = \sigma(\mathsf{X}_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\text{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$

Preface

In this course notes, we introduce the operating system. ECE391 is known as one of the most difficult undergraduate courses in UIUC, not just ECE department. It's a crazy course that consumes up to several hours a day and is really an incredible undertaking.

As you may have guessed, the course director is Prof. Steve Lumetta, a hacker, teacher, and giant. But as you know, his notes is hard to understand if you're not a native English speaker, because he uses lots of *big words* and terminology. His notes is better for reading for several times and get a strict understanding of the concepts.

This notes is created with a different perspective. As I'm a student, I care about how we learn the the course, and how the knowledge incrementally builds up. Thus, I suggest you to use this notes as something you can turn to after you find that you can't understand something talked in the course, but not something you can use from the scratch. Please attend courses, otherwise even if this notes is super correct, you can't make sense out of it. Again, don't just look into the notes - please use official course materials, and the teaching assistants are really willing to answer your questions.

This notes also include communicative computer networking stuff because my final project is about computer networking. One of the best thing about this course is its final project - you can build anything from scratch. If you also want to build a network stack from scratch, please do read the second chapter about communicative networks.

Disclaimer

This notes is not strict, because it was written by a student who attended the course. If you see a similar question in the exam, you should use what's on the course to answer. I'm not responsible for any of your score, but I'm happy to discuss technical questions on GitHub. Please open an issue if you would like to discuss.

Please again note that this is NOT an official study material for ECE391, but something like a helper tool when you learn it. If you find anything wrong, please contact me and we can discuss how we can fix it.

Some figures are taken from Chinese resources so they have Chinese. Please translate them if you would like to. Some figures are taken from the official course materials, please contact me to re-draw or delete them if you're a course staff and you've consulted the professor.

Chapter 1

Operating Systems

1.1 The Linux Kernel

The kernel is the set of softwares which are closest to the hardwares in a machine.[1] Any operating systems has a kernel, including Windows, Mac, Android, iOS, and of course, Linux - as shown in Figure 1.1. Windows uses an NT Kernel, OS X and iOS uses the Darwin Kernel, and Linux and Android uses the Linux Kernel.

As this paper concerns mostly about Linux, we take Linux kernel to illustrate the ideas. Essentially, everything that you do on the screen is about kernel. For example, you open a software, then the software will raise a system call to allocate some memory, which is handled by kernel. When you want to compile the source code you've just finished, you still need the compiler to ask kernel to compile it. In this case, kernel is about the kernel of all applications you do on your computer.

1.1.1 User Space and Kernel Space

When you turn on your computer, the system goes into privilege level 0 (real mode). The kernel then configure stuff like `IA32_LSTAR MSR` register with an address through the `WRMSR` (write MSR) instruction. After that, kernel drops the privilege from level 1 to level 3 to enter the user mode (protected mode, when A20 address line is activated), like shown in the code block below.[2]

```
1 // setup hardware..
2 WRMSR IA32_LSTAR = 0x12236A00;
```

[1] https://www.youtube.com/watch?v=mycVSMyShk8
[2] https://www.youtube.com/watch?v=fLS99zJDHOc

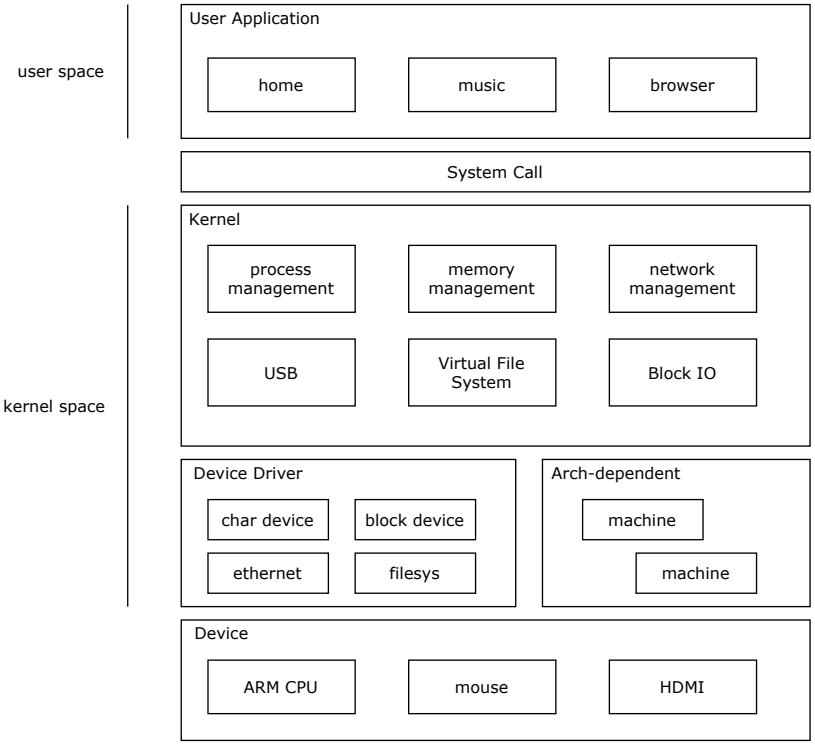

Figure 1.1: The Linux Kernel Architecture.

```
3 // enter user mode (privilege level 3)
```

Now the hardware is configured down - we cannot reconfigure this register(and hardware) anymore. If we want to go to the privilege level 0, we must use a **syscall** (system call). However, we cannot control what will be executed after we enter privilege level 0, because the address of the syscall is fixed. A syscall is like a bunch of commands hard-coded into the Linux kernel, and we can only call it with the whole bunch.

A question is, what is the difference between kernel code and user code? There is no difference in the code. Kernel code is the code to be executed in the kernel mode (kernel space), and user code is the code to be executed in the user mode (user space).

How about the data transfer between user space and kernel space? *Halo Linux*

Services gives an answer.[3] The kernel often needs to copy data from userspace to kernel space; for example, when lengthy data structures are passed indirectly in system calls by means of pointers. There is a similar need to write data in the reverse direction from kernel space to userspace. This can **not** be done simply by passing and de-referencing pointers for two reasons. First, userspace programs must not access kernel addresses; and second, there is no guarantee that a virtual page belonging to a pointer from userspace is really associated with a physical page. The kernel therefore provides several standard functions to cater for these special situations when data are exchanged between kernel space and userspace.

This cannot be done simply by passing and de-referencing pointers for two reasons. First, userspace programs **must not** access kernel addresses; and second, there is no guarantee that a virtual page[4] belonging to a pointer from userspace is really associated with a physical page. The kernel therefore provides several standard functions to cater for these special situations when data are exchanged between kernel space and userspace.

For a more detailed explanation of the functions `copy_to_user` and `copy_from_user`, according to Caf[5], is that:

- They check if the supplied userspace block is entirely within the user portion of the address space (`access_ok()`) - this prevents userspace applications from asking the kernel to read/write kernel addresses
- They return an error if any of the addresses are inaccessible, allowing the error to be returned to userspace (`EFAULT`) instead of crashing the kernel (this is implemented by special co-operation with the page fault handler, which specifically can detect when a fault occurs in one of the user memory access functions)
- They allow architecture-specific magic, for example to ensure consistency on architectures with virtually-tagged caches, to disable protections like SMAP or to switch address spaces on architectures with separate user/kernel address spaces like S/390

[3] https://www.halolinux.us/kernel-architecture/copying-data-between-kernel-and-userspace.html

[4] A virtual page connects to a larger system that relates to virtual memory. This space exists as a virtual storage on an operating system that can be transferred to a physical memory storage known as the process of paging. Within the operating system, only virtual memory can be stored whereas the transfer that takes place towards a physical memory storage refers to the hardware that holds the transmitted data.

[5] https://stackoverflow.com/questions/12666493/why-do-you-have-to-use-copy-to-user-copy-from-user-to-access-user-space-from

Now consider such a questions that[6], you want to transfer a linked list of structs from the user space to the kernel space. In this manner, whether you are reading or writing from the user space, you both need to make a local copy of the struct using `copy_from_user` in the kernel code in order to use it. If you're reading, you can save the local copy on the kernel stack or in the kernel memory (heap) in order to avoid direct accessing the user space memory. If you're writing you need to firstly get the value locally then move the data back without direct accessing as well.

This is because the memory addressing methodologies inside these two spaces are different, as mentioned in Section 1.4.4. The memory in the kernel space is not converted to virtual memory while it is in user space. With the Memory Mapping Unit (MMU) it's fairly easy from kernel to get confused where the exact address is in the user space. A graphical illustration is shown in Figure 1.2 to start the discussion in the context of MP1.

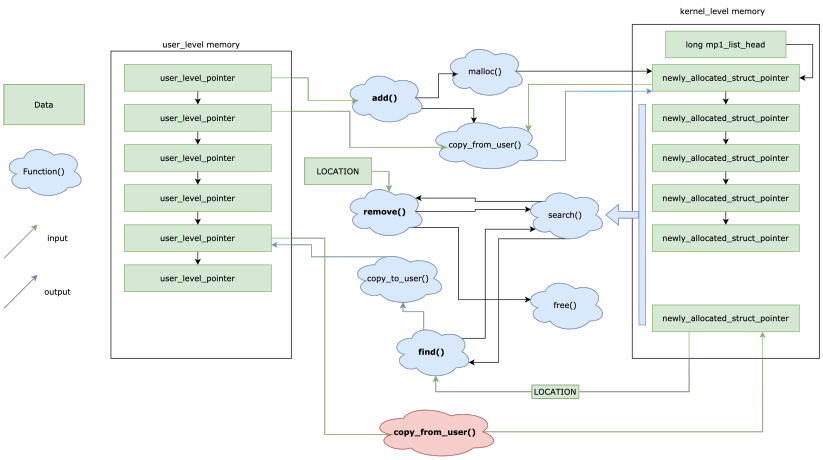

Figure 1.2: The memory layout in the kernel space and user space in MP1. The crucial operation in this scenario is marked with red color. Note that there is never a line connecting a function that modifies memory with the user space memory. If the code is executed in kernel, it has only the priority to modify the kernel memory without using MMU.

The first function is the `add()` function. It takes a pointer to a struct in the user space as argument, and aims to add this struct into the head of the linked list of struct in the kernel space memory. To achieve this, it calls `malloc()` to allocate

[6]Actually, this is the scenario happened in MP1 of UIUC-ECE391 Course (Operating Systems).

a bunch of space in the kernel space memory, and then use `copy_from_user()` to copy the whole struct pointed by the argument pointer to the allocated memory address in kernel.

The second function is the `remove()` function. It accepts a `LOCATION` (which is a field in the struct) integer and search in the kernel space to find the struct with the same value. After getting the pointer to that struct the function calls `free()` to free that memory.

The last function is `find()`. It accepts a pointer to a user-level struct, just like `add()`, but it only needs the `LOCATION` field of that struct and uses it to find the struct in the kernel space struct. After finding the pointer to the kernel space struct, it copies the whole struct in the kernel space and calls `copy_to_user` and returns. The trick in this function is the extra `copy_from_user()` function, which is cricial to the function to behavior normally. Without this extra function, the kernel is going to access the user space memory directly and try to get the `LOCATION` field in the user space, which is an extremely dangerous dereferencing operation.

A very interesting question is, then, why the operating system needs both user mode and kernel mode? If it's so complex, why not just make the everything simple and use just one memory space? According to GeeksForGeeks[7], anything related to Process management, IO hardware management, and Memory management requires process to execute in Kernel mode, which means that basically all system calls (`syscall`) needs to enter the kernel mode for execution. For a graphical illustration please refer to Figure 1.3.

A further question is, for example, when using GDB, why all the disassembled data starts from one address? This does not make sense because it may overwrite other existing data. Well, this leads us to the concept of MMU.

1.1.2 Descriptor Tables

There are 3 kinds of descriptor tables: GDT (Global Descriptor Table), IDT (Interrupt Descriptor Table), and LDT.

GDT, as implied by its name, is global, or gloablly visible. In other words, all programs are able to access GDT. The **segment selector** has a 13-bit index number, which stores the index of the segment in GDT. This means GDT can contain up to 8192 elements. Moreover, GDT is nothing more than an array allocated by the operating system for specific use (instead of general use).

[7] https://www.geeksforgeeks.org/user-mode-and-kernel-mode-switching/

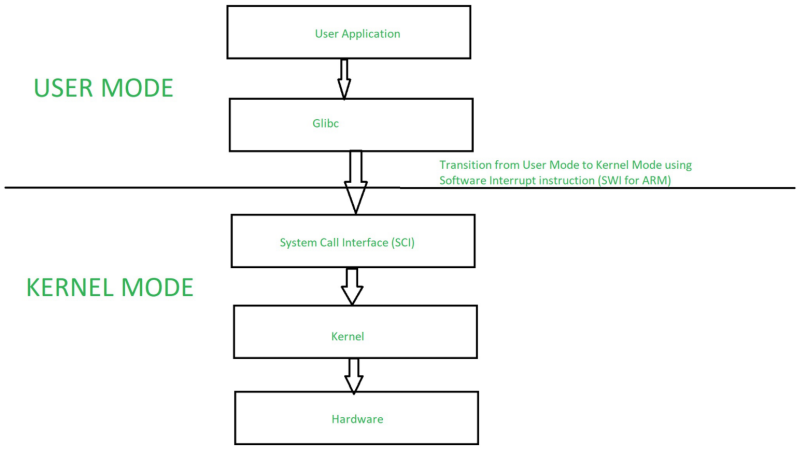

Figure 1.3: The progress of a system call. This call is conducted by user code and drive the kernel by making a `syscall`. Note that `glibc` is the GNU Standard C library designed for Unix-like operating systems. The figure has credits for *GeeksForGeeks*.

15 3	2	1 0
索引号	T1=0, GDT T1=1, LDT	请求特权级别

Figure 1.4: Layout of the segment selector.

After using the segment selector we get the segment descriptor, which is stored in GDT. We see the elements stored in the GDT array are actually segment descriptors. The **segment descriptor** (not file descriptor) will describe all information about the segment of a program, as shown in Figure 1.5.

63 56	55	54	53	52	51 48	47	46 45	44	43 40	39 32
	G	D	0	0		P	DPL	1	type	
Base address 31 : 24	flags				Limit 19 : 16	access				Base address 23 : 16
31 16						15 0				
Base address 15 : 0						Limit 15 : 0				

Figure 1.5: Layout of the segment descriptor.

IDT, as implied by its name, is a table used to store **interrupt descriptors**. The interrupt descriptor is used to describe all information about an interrupt, as shown in Figure 1.6. An interrupt is a process executed every time when there is a interrupt signal sent from a device (instead of CPU). The IDT table stores not only interrupts but also exception and system calls.

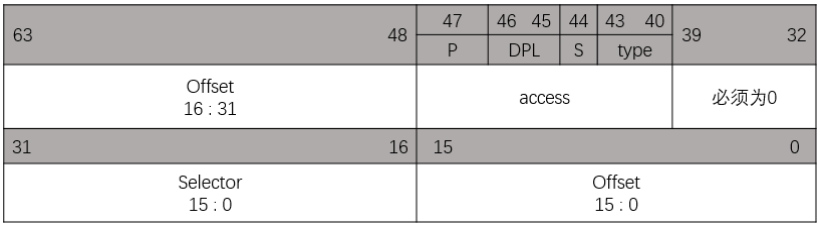

Figure 1.6: Layout of the interrupt descriptor.

1.1.3 Interrupts

When an interrupt is sent from the device, the PIC (Programmable Interrupt Chip) chip receives the signal and invokes a signal to CPU. Then the CPU leads the user program to an **assembly linkage**, which is essentially a wrapper of the interrupt handler. In other words, the assembly linkage firstly executes some flag storation, then call the interrupt handler, and after the handler function it restores the flags, in order to maintain C calling convention and return from interrupt. The **interrupt handler** is a function executed when the corresponding interrupt signal is received from the device.

Note that at the very end of an interrupt handler, an `IRET` command is executed instead of `retern`, because we're returning from kernel space (interrup handler environment) to user space (program running). The `IRET` command performs a context switching including the stack switching and privilege switching. The detailed popped information from the stack is shown in Figure 1.7.

The installation of an interrupt handler is completed in the function `request_irq`. This function is located in the device, and basically links the device with an IRQ number, and register the number so the CPU knows about it.

There are 2 kinds of gates - TRAP game and INT gate. The TRAP gate does not clear IF by default, while the INT gate does. Thus, the trap gate is useful for software interrupts, while the int gate is useful in hardware interrupts. The hardware interrupts are raised when a device sends a signal to the CPU saying

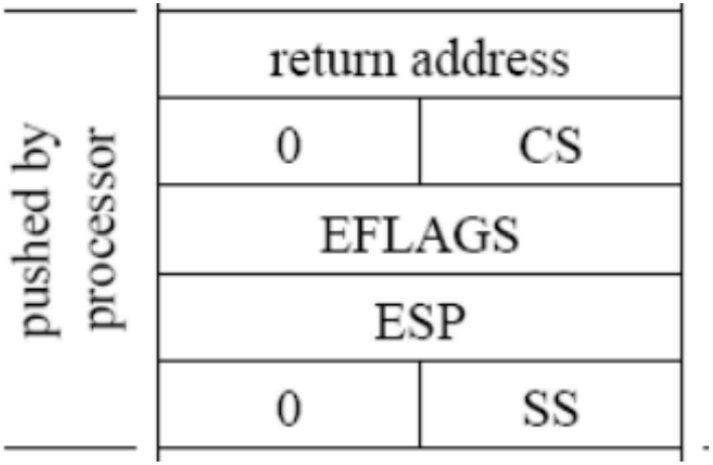

Figure 1.7: The values to be popped from the kernel stack in a context switching process. **CS**: code segment. **SS**: stack segment.

there should be an interrupt, while software interrupts[8] includes system calls and exceptions, which does not contain any operations with hardware devices. The priority of a software interrupt is between the hardware interrupt and the user-space program, such as printing a string to the screen and raising an segmentation fault exception.

1.2 System Calls

The system call is a special kind of interrupt. In the Linux kernel, it's occupying the `0x80` entry in the Interrupt Descriptor Table (IDT). Just like other interrupts, the system call will also trap the user program into kernel mode, and return from the kernel mode to user mode after the system call is finished. However, system call is a kind of software interrupt, because it's raised by a program running on CPU instead of a interrupt signal delivered from external devices.

System call has several types, and the most common ones are listed in the table below. The first two system calls are used for executing the syscall utility

[8]Some literature also calls software interrups **tasklets** because it's a small and quick task for the operating system, just like what we wrote in MP1.

functions, while the second last four system calls are used for file operations.

Command	Arguments
`halt`	`uint8_t status`
`execute`	`const uint8_t* command`
`read`	`int32_t fd, void* buf, int32_t nbytes`
`write`	`int32_t fd, const void* buf, int32_t nbytes`
`open`	`const uint8_t* filename`
`close`	`int32_t fd`

Table 1.1: The most common system calls in kernel.

For the next separate parts about the system calls, please compare to Figure 1.8 for reference. When the computer is booted (power on), the operating system calls `system_execute("shell")`, which launches the very first terminal.[9] In some modern operating systems, if we kill the very first shell, the whole computer system is going to halt. But in Machine Problem 3 in ECE391, UIUC, the very first shell is designed to be permanent - if you halt the very first shell, it will be launched again immediately and automatically.

After the `system_execute` kernel function is launched, the EIP register in CPU is going to point to the user space program. In other words, now the CPU is running in user mode. At this moment, the familiar terminal interface is already displayed on the screen and should be visible. A prompt should appear, like `jack root/ece391>` followed by your commands.

In the user mode, the user (you) can launch any program of concern. However, almost all user-level programs make system calls, like `read`, `write`, `execute`, `halt`, etc. For example, when you type in `ls` (the user-space program), you'll trigger the system call `system_execute("ls")`, which prepares the actual `ls` program. At the very end of the `ls` program, the halting function `system_halt()` is executed to halt the program.

The actual system call is done by the command `INT 0x80`, which is marked with order number 3 in Figure 1.8. When `system_executable` is executed, to trap the user program into the peogram, the code actually sends a software interrupt to

[9]Note that the shell is different from the terminal: terminal is the user-visible operator space that controls the user-invisible shell program. In other words, we can use multiple terminals to change one shell; we can also use one terminal, in this terminal we launch a shell using the current shell, and thus shift from one shell program to another.

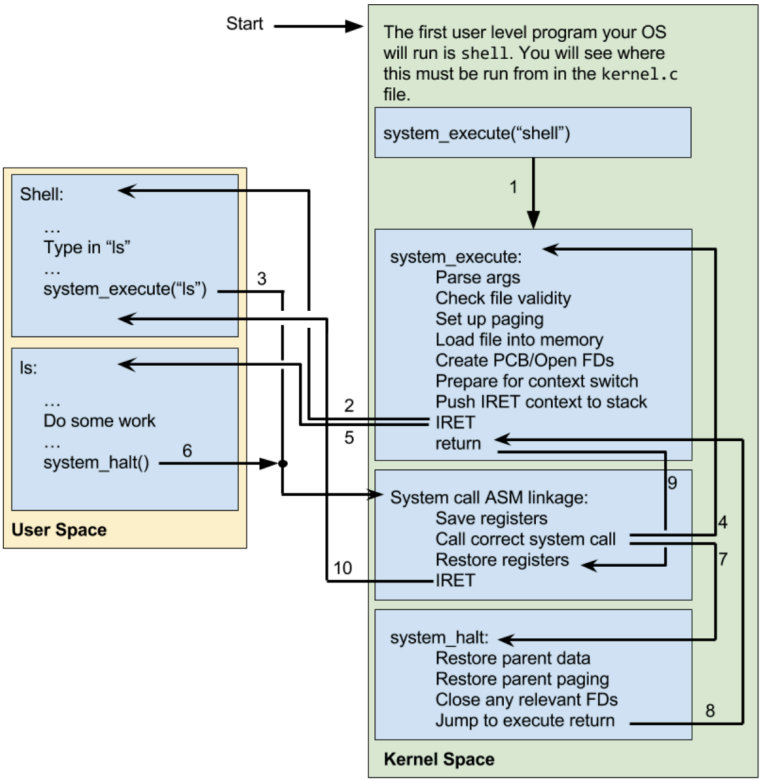

Figure 1.8: The overview of the `execute` and `halt` system call workflow.

the CPU by using command `INT 0x80`. When `INT 0x80` is executed, the program goes to IDT and will be dispatched to the assembly linkage. The system halt is exactly the same, as noted by number 6 in Figure 1.8.

For a deeper understanding of the system call, we'll go through all the most commonly used system calls below.

1.2.1 execute

The `execute` system call is the program to bootstrap the actual user program. After the kernel space program executes the command `system_execute`, or after the assembly linkage calls the subroutine `system_execute`, the arguments sent in will be parsed. For example, when the command `cat jack.txt` is passed in, it will be parsed as 2 arguments `cat` as the command name, and `jack.txt` as the

first (and only) argument name.

Afrer parsing, the program checks whether the command and arguments are valid. A typical wrong command would be a bad command that does not exist or the number of arguments does not meet the specs of the command.

If the system call is valid, the function goes into the utility part by first setting up paging. The setup is pretty straight-forward for a single process system: just map a randon virtual address to the ending kernel address of the kernel space. In other words, the new user program memory is "allocated" just after the kernel memory. As discussed, the kernel memory takes a large page (4MB), and here we design each user program takes a large page (4MB) as well. Note that the user stack and user heap are allocated in the user space memory.

After the paging is set up, we need to load the program file into memory. For example, if we launch the user program `shell`, we firstly examine the executable by looking at its magic number. If the magic number is valid, we copy the content of the whole executable file `shell` into somewhere in the user program memory for later execution.

Then we create the process control block that describes the current process. We initialize a file descriptor table for it, which contains two default files `stdin` and `stdout`. The `stdin` file is used to pass characters from the visible terminal to the invisible buffer, while the `stdout` file is used to pass characters from the buffer to the terminal. Thus, `stdin` must have access to the file read function `terminal_read` and `stdout` must have access to the file write function `terminal_write`.

After that, the kernel should prepare to go back to the user space and give the access back to the user. To achieve that, the kernel program should update the TSS segments to store the (address of) kernel stack, as described in Section 1.6.2. Other operations may be required to assure all the kernel information has been saved so the user program can go back to the kernel normally.

At last, we push where we want to go to user program by pushing the 5 registers onto the kernel stack, as illustrated in Figure 1.8. Then we call the `IRET` command to return the terminal access to the user.

1.2.2 halt

The `halt` system call is used to end a system call. Because it's a system call, we can't call it in the kernel space, which means that it must not be a subroutine in the `execute` system call. Instead, it's always executed at the very end of a user program.

After the `halt` system call is executed, as same as `execute`, it firstly goes to the assembly linkage to call the utility function `system_halt` in the kernel space. This function unravel the current process, and restore all information in the parent process, including parent data, parent paging, and parent TSS.

The tricky part is the last command by `system_halt`: it jumps to execute `return` in the `system_execute` function, instead of returning back to `IRET` in the assembly linkage immediately. The reason is that the user program that calls `system_halt()` is `ls`, which means if it returns directly, EIP will go below `system_halt()` in the user program `ls` instead of the command `system_execute("ls")` in user program `shell`.

1.2.3 read, write, open, close

The most commonly used file system calls include `read`, `write`, `open`, `close`. As they're all wrapper functions of those in the file system, we only take `write` as an example.

As all system calls are called by the user program, all the file operation system calls are also triggered by the assembly linkage `INT 0x80`. For example, in the user program `vim`, a function `user_write` is called. This function traps the program using `INT 0x80`, thus leading the program to kernel space.

After CPU gives the access to the kernel, it enters the system call body function `write`, with arguments `fd` as the file descriptor, `buf` as the buffer storing the content to be written into the file, and `nbytes` indicator limiting the length of the buffer.

The `write` function then checks if the buffer is valid, whether the file descriptor has been set up in the file descriptor array,[10] and check if the file operation dispatch function in the file descriptor array is successfully set up. If all above passes, the syscall `write` calls the writing utility function[11], like `terminal_write` for the `stdin` file.

After the utility function returns, the syscall `write` returns as well, back to the next instruction of the `INT 0x80` command.

[10] If the file descriptor is already in the array, it means that the requested file is successfully setup, so it's okay for a `write` system call.

[11] The utility function means the function that does the real work. The counterpart of a utility function is a wrapper function.

1.3 Signals

The **signal** is another type of soft interrupt, which is generated by softwares instead of hardware devices. Sometimes some behaviors of a program interrupts the program itself, and sometimes some certain combined keystrokes raises an interrupt. For example, the `sigsegv` signal is generated by a program when there is a **segmentation fault** in the user-space program, and the program raises this interrupt in order to force to kernel to stop executing this user program.

There are two important signals that can't be ignored as Non-Maskable Interrupts (NMIs): `sigkill` that terminates the process, and `sigstop` that stop a process.

The signals are firstly called by user space programs, but generated and handled by the kernel.[12] For example, for the signal `sigkill`, it is called when the user program calls the function `kill()`, while this function finally redirects to the function `send_sig_info()` in the kernel, which generates the signal.

1.3.1 Hardware Triggers: Terminal Keystroke

As mentioned, the signals are software interrupts, but we observed that when we send some keystroke combinations to the terminal, it generates signals as well. Why is this the case?

The reason is that the keyboard strokes firstly generates hardware interrupts, then using the interrupt combinations to produce a signal. For example, after the user sends the `^c` keystrokes to the terminal, the process of pressing the two keys is essentially a hardware interrupt. As usual, the hardware interrupt prevents the CPU from executing the user space program, and the CPU now enters the kernel mode.

The terminal sees that there is a `^c` combination carried by the hardware interrupt, so it interprets this hardware interrupt as a `sigint` signal, and records the signal in the PCB of the process.

Before the program counter wants to go back to the user space from the kernel space, it looks for signals in the PCB. Now as the CPU captures a signal registered in the PCB, it goes to execute this signal first. The default behavior of this signal is to terminal the process, so the kernel terminates the whole process and does not go back to the user space any more.

Note that only normal processes in the terminal can receive such signals, and

[12] https://blog.csdn.net/modi000/article/details/104672983

unblocking processes, background processes, and daemon processes cannot get any of the signals using keystrokes. The explanation of these processes will be explained in Section 1.6.8.

1.3.2 Software Triggers: System Functions

To trigger a software interrupt using software (code), we need to call the trigger functions defined by the operating system. For Linux, there are 3 functions defined by the operating system: `kill`, `abort` and `alarm`.

The `kill` function accepts two arguments: the process ID and the signal ID. When you call this function in your code, the program will send a signal to the kernel to terminate the target program. When your program does not have the privilege to do so, this function returns `-1` as sets the error code.

The `abort` function sends a `sigabort` signal to the kernel without any arguments. This function automatically detects the function who calls it and sends the `sigabort` signal to the kernel.

The `alarm` function has an argument of number of sections. After the elapse of this amount, the kernel will send the a `sigalrm` signal to the process, which by default terminates the current process.

1.3.3 Core Dump

The **core dump** happens when the kernel receives a `sigsegv`, `sigbus`, `sigsys` signal, etc. These signal indicate that there is something wrong with the user space program that may potentially crash the kernel, so they raise a signal to stop the program. For example, the `sigsegv` signal indicates that there is an invalid memory reference/dereference, and `sigsys` means a bad syscall.

The **core** is a file storing the memory image (snapshot) just before the program crashes, just like the black box of a plane, which is mainly for debugging and finding what's wrong with the program. The `core` file can be analyzed by gdb, with the stack back-tracing and pointer value functionalities.[13]. The details are illustrated in the blog as shown in the footnote.

[13]https://blog.csdn.net/qq_39759656/article/details/82858101

1.3.4 Signal Workflow

The signal action struct is shown in Figure 1.9. When the user program receives the signal, it enters the kernel space and calls the `do_handler()` function. This function ends up with setting up the user mode stack and forces the kernel to go back to the user space, because the signal handler is in the user code. The user mode stack also includes the `sigreturn()` system call function binary so that the user program can go back to the kernel again and call the `system_call()` function in the kern el space. After this function returns, the program enters the user space once again and goes on with the normal user program flow.

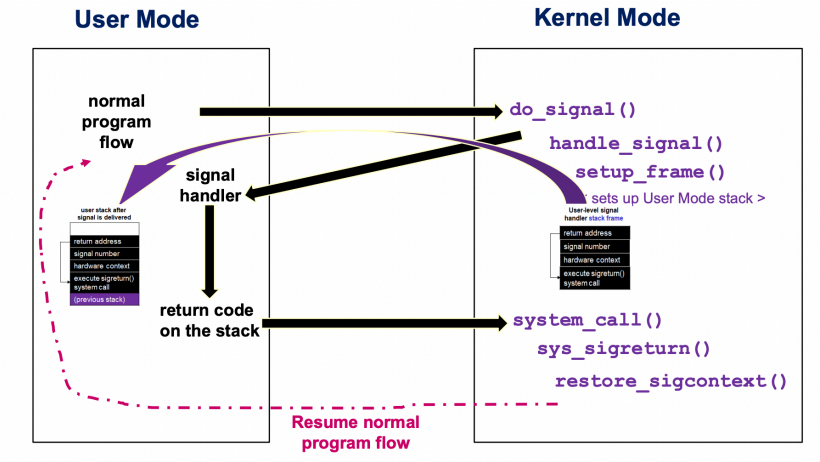

Figure 1.9: The workflow of a signal.

There is a security concern here, though. Because the executable binary is on the stack, the hackers can penetrate the system by replacing the binary with malicious programs. The developers fixed this issue by changing the binary to a pointer that goes to a well-defined space in memory, so that the hacker cannot directly get access to the binary code.

1.4 Memory Paging and Segmentation

The memory paging system in Linux is one of the most tricky part. In this section, we mainly discuss the memory paging and memory segmentation techniques and principles, with Linux as the working example.

1.4.1 Memory Address Types

There are 3 memory address types: physical address, linear address, and logical address.

The **physical address** is what's in the memory hardware. As discussed in Section **??**, there are various kinds of memories, and they all provide interfaces to the operating system. The operating system makes use of the interfaces provided by the hardware to address the RAM units in the memory, and thus achieve read and write functionalities.

The **linear address**, also called the **virtual address**, is an intermediate layer between physical address and logical address. As illustrated in Figure 1.10, the offset in the linear address and logical address is the same. In other words, the logical address contain the information *including* the base address and offset, while we can use the base address and offset to calculate the linear address.

The **logical address**, also called the **relative address**, is the combination of a segment selector and the relative most off in that segment. In a userspace program, the address we write in the code is actually a logical address, which will be translated to a linear address first, and then translated to a physical address. For example, given the logical address `0A5Ch`, we calculate its binary equivalence `0000 1010 0101 1100b`. If the page size is 1KB (2^{10}B), then the offset should be `10 0101 1100b` because we take the last 10 bits. Similarly, the page number is then `0000 10b` (the beginning bits), which is equivalent to `2d`. We then use the page number to find the block number (if it exists - say it's `4d`), then the answer of physical address should be `0001 0010 0101 1100b`.

Here we introduced a conpect called **block**. A block is simply a page or part of a page on the physical device, which is not so important, because it's handled by the device. For programmers, we only need to consider about the paging mechanisms.

Note that without MMU, the linear address will be sent to the device directly, so the linear address becomes the same as physical address. In other words, MMU is in charge of translating the linear address to physical address. This also indicates that MMU has nothing to do with translating the logical address to the linear (virtual) address. Details will be discussed in 1.4.4.

1.4.2 Memory Paging

Before there was paging (i.e. before CPU 80386 was invented), all the memory addresses were real, so it's called **real mode**. In real mode, all memory addresses

are accessible by both kernel and user applications. This is caused by the fact that there is no virtualization in memory, and all memory is physical.

This approach is extremely dangerous from the perspective of memory security. With user space program able to access kernel memory, the operating system is pretty easy to crash and even attacked by a hacker.

Consequently, the **protected mode** is invented to split the memory of user space and kernel space. In the protected mode, there are two modes for managing memory: segment mode and segment/paging mode. The segment mode means that programs use merely segments to address, whiel segment/paging mode means that programs use not only segments but also paging to address.

The **segment** for a program is used to guarantee the security of the kernel space. The segment system uses a **Global Descriptor Table** (GDT) to accomplish this goal. Each descriptor in GDT contains all the basic attributes of a memory segment, including segment base, segment boundaries (segment limit), and memory type.

Using the GDT, each segment selector (which is a part of the logical address in the memory) corresponds to a segemnt descriptor in GDT. Then the segment descriptor gives all the information to find the linear address in the memory. Note that the offset in the linear address is the same as that in the logical address. The usage of GDT is illustrated in Figure 1.10.

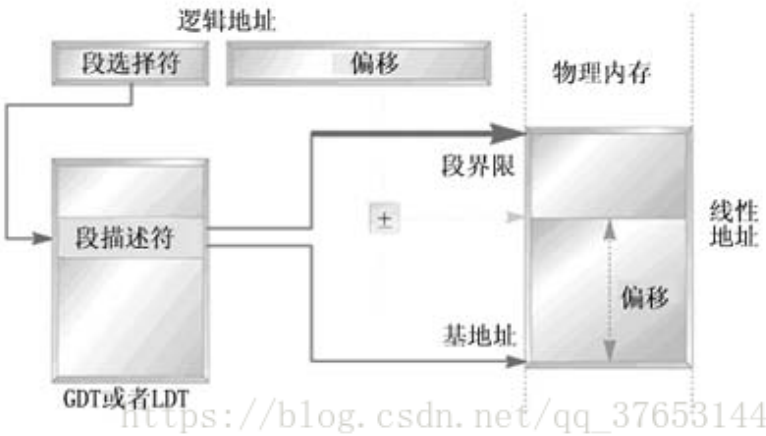

Figure 1.10: Rule of indirectly memory accessing using GDT.

Please note the difference between a page and a **page frame**. Consider such a

computer with 256M physical address and 32-bit addressability (4G). This means the physical memory space is 256M, while the virtual memory address is 4G. In this case, when a program requires 4G memory to run, it cannot be loaded into the memory in one time. Instead, it must have an **external** storage device (like FLASH memory, as mentioned in Section **??**) to store the run-time environment. If we assume the page and page frame size is 4K (they must be the same), then there are 1M pages (in the virtual memory) and 64K page frames (in the physical memory).

As mentioned in Section 1.4.1, MMU is used to translate linear address to physical address through the paging technique. The translation from linear address to physical address is shiwn in Figure 1.11.

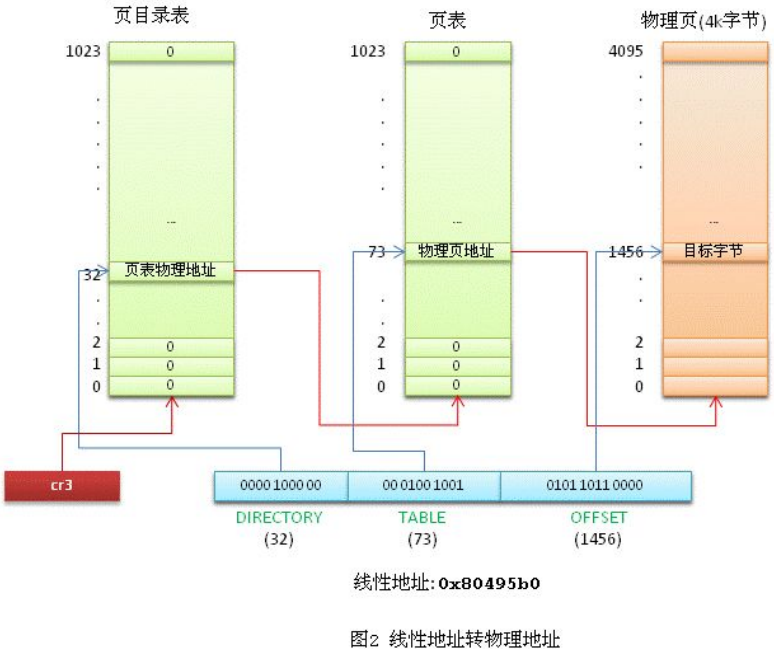

图2 线性地址转物理地址

Figure 1.11: The process to translate linear address to physical address.

Before using the linear address, it should be noticed that there is a **cr3** register. This register is used to store the physical base address (physical starting point) of the page directory table. The highest 10 bits are page directory table offset, the following 10 bits are page table offset, and the last 12 bits are physical page offset.

Assume the linear address to be translated is `0x80495b0` or `0000 1000 0000 0100 1001 0101 1011 0000`. Clearly the page directory table offset is `0000 1000 00`, so the CPU is going to look for the 32th item in the page directory table, which has been pointed to by cr3. Then using the address stored in the page diretory we get the page table address, and we have the page table offset `00 0100 1001` so we get the page address. Using the physical page address we now get the target address in the physical memory.

According to Figure 1.12, we only need 20 bits in PDE for the address of page table, because the memory is aligned in physical memory. For example, the base address of a page table can only be `0x1000, 0x2000...`. This is because the memory needed for a page table is 4KB, and due to the mandatory alignment in memory, the least 12 bits must be 0, thus the base address must be a multiple of `0x1000`. Using this idea, for a 4MB page defined by a PDE, it must be a multiple of `0x40 0000`, because the 22 least significant bits must be 0, and `2` in the PDE means `0x80 0000`. Similarly, a 4KB page defined by a PTE must be a multiple of `0x1000` as well, as each of them takes 4KB.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame								Reserved (must be 0)				Bits 39:32 of address[2]				PAT	Ignored						G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored				G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Figure 1.12: The Intel Manual for IA-32 on the paging-related registers and paging structure entries.

1.4.3 Comparison: Paging and Segmentation

In Linux, there's no difference between logical address and linear address, because all segments in Linux start from `0x0`. In other words, because the base address for all the segments is `0x0`, there's no need for the logical address to be translated into an intermediate (linear address). A detailed graphical illustration is shown in Figure 1.13.

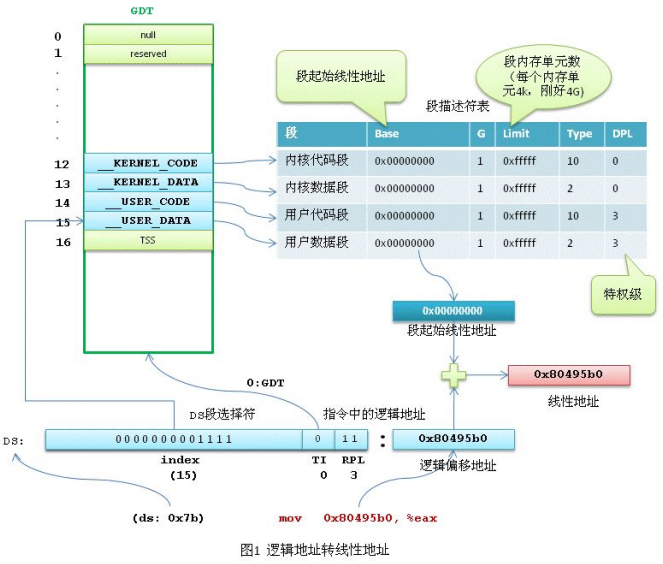

图1 逻辑地址转线性地址

Figure 1.13: Segment system from logical address to linear address in Linux.

In the illustration, we have four descriptors: `__KERNEL_CS` and `__KERNEL_DS` for kernel program, and `__USER_CS` and `__USER_DS` for user program. Note that all the Linux programs can be divided into these four groups, so there's no need to implement a segment descriptor for each program specifically anymore - the logical address will refer to one of the four instead.

Segment	Base	Limit	Type	DPL
`__KERNEL_CS`	0	4GB	10	0
`__KERNEL_DS`	0	4GB	2	0
`__USER_CS`	0	4GB	10	3
`__USER_DS`	0	4GB	2	3

Table 1.2: The properties of the four segments in Linux Memory System.

In IA-32 CPUs, the privilege level of an executed program is determined by the privilege level of the code segment in which the program is executed. Recall that a program counter for IA-32 CPUs consists of a segment, specified by the CS segment register, and an offset into the segment, the EIP register. The privilege level of the code segment then is determined by its segment descriptor. A segment descriptor has a field for specifying the privilege level of the segment.[14] Thus, as Linux makes all code/data starting from the same base, EIP always starts from the same address `0x0`, so we can simply ignore segmentation base here. However, segmentation still operates because it offers different type and descriptor privilege level (DPL) for each program, so it cannot be totally omitted.

The selection of paging and segmentation is somewhat a trade-off between internal fragmentation and external segmentation, and it's hard to tell which is better. The only thing to be certain is that certain tools are useful in certain situations.

Another typical trade-off about paging is the page size, which is a trade-off between fragmentation and time. If the page size is too large, like 1GB, then the large fragmentation would cause lots of problems. On the other hand, if each page is only 4B, although the fragmentation is now reduced, we need lots of storage for the page tables.

In modern operating systems, segementation and paging are both used. The physical memory is firstly divided into segments, which are mostly 4GB large, and then divide each of which into pages for detailed dereferencing, thus constructing a coarse-to-fine approach. To solve the fragmentation and the "big segment" problems inside each segment, paging is used. With paging a larger segment is broken into several pages and **only the necessary pages remains in physical memory** (and more pages could be allocated in memory if needed). Now the question comes: why programmers always see a segment fault instead of page fault in the when programming in C, Pyhotn, and almost all others? Firstly, as mentioned in

[14] https://www.linuxjournal.com/article/6516

Section 1.4.1, the segments are divided into four in Linux, including the `_KERNEL` two and the `_USER` two. Thus, all mechanisms in the last paragraph occurs, but Linux use the same base (0) for all segments and, since the page tables are individual per process (with 4GB virtual memory space), a page fault is difficult to happen.[15] On the other hand, as segmentations are managable by users, they're more likely to show an error, which leads to the result that programmers see `Segmentation Fault` a lot but seldom see `Page Fault`. Secondly, in most cases, the OS will handle the page fault **internally** and restart the process from its current location as if nothing happened. For example, guard pages are placed at the bottom of the stack to allow the stack to grow on demand up to a limit, instead of preallocating a large amount of memory for the stack. Similar mechanisms are used for achieving copy-on-write memory.[16] In other words, page fault is handled by the Operating System, and modern OS already obtain the capability to handle them internally (transparent to programmers), while segment fault cannot be handled by the operating system - it has to be solve by the programmer. Thirdly, for those really causing paging faults in the operating system, with protection and indirection, the error message shown to users is transferred to a segmentation fault to form a uniform interface. For example, if the programmer tries to dereference the NULL pointer, the operating system detects it but also converts it to a segment fault and follow it with a detailed error message showing that the user dereferenced a NULL pointer. Thus, although dereferencing the NULL pointer is a page fault per se, the user space program receives a segmentation fault error message and already know what to do, so he does not bother to learn what a page fault is, what does it mean, what is an operating system, how to fix it, and in the end, why he even need to know all these as a data analyzer.

1.4.4 The MMU and TLB System

The memory mapping unit (MMU) is a module that translates virtual memory to physical memory using the page table system and a bunch of TLBs (Translation Lookaside Buffers). MMU is usually located in CPU, but sometimes also located in an Integrated Chip (IC), while the location of TLB is more flexible, maybe between CPU and CPU cache, maybe between CPU cache and primary storage memory, or between multi-level caches. The MMU's obligation is to translate the logical memory into physical memory using the page tables, and at the same

[15] https://stackoverflow.com/questions/24358105/do-modern-oss-use-paging-and-segmentation

[16] https://stackoverflow.com/questions/12645647/what-happens-in-os-when-we-dereference-a-null-pointer-in-c

time the MMU stores the result (a pair of logical address and physical address) into TLB. Thus, the TLB becomes a cache for the translation results. In this manner, when another translation happens, MMU firstly turns to TLB and look for whether it contains the translation it wants, if it does not, it then turns to MMU for translation. Note that MMU does **not** locate logically - it's a physical device on the motherboard because the software approach would be too slow for a translation. Also, TLB does not locate in MMU - it is an independent cache. The system is illustrated in Figure 1.14. There are usually multiple TLB

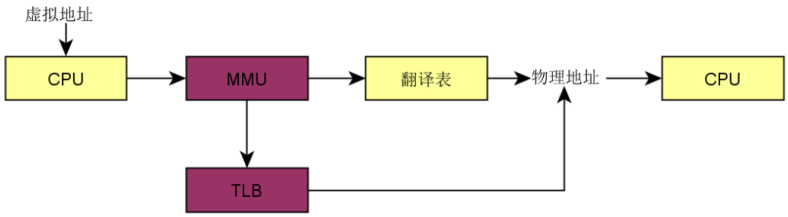

Figure 1.14: The MMU/TLB system working rules.

buffers stored. For example, after translating the paging `0x001000-0x0B2000`, MMU stores the translation into one TLB. Then another translation is done, say `0x002000-0x0C2000`, MMU stores into TLB again as another entry. Then each one of the two translation queries happens, MMU would be able to find the result **immediately** because they're both stored, thus saving lots of time. Concerning the characteristics of TLB and continuous paging access, there is a programming trick underlied, which is often omitted by programmers these days. Consider the code below:

```
1 int x, y;
2 result = 0;
3 for (x = 0; x < 1024; x++) {
4   for (y = 0; y < 4; y++) {
5     result += matrix[0][x] * matrix[y][x];
6   }
7 }
```

The array storage should be continuous in memory (as shown in Table 1.4). From the table we can also draw the conclusion that each entry is 32-bit, because it takes an address space of 4, and x86 is byte-addressable.

Virtual Address	Data
0xc80000	matrix[0][0]
0xc80004	matrix[0][1]
0xc80008	matrix[0][2]
...	...
0xc81000	matrix[1][0]
0xc81004	matrix[1][1]
...	...

Table 1.3: The virtual data address in memory.

In the code snippet above, we're going through the matrix in such a way:

Step	Data	Virtual Address
0	[0,0], [0,0]	0xc80 000
1	[0,0], [1,0]	0xc80 000 | 0xc81 000
2	[0,0], [2,0]	0xc80 000 | 0xc82 000
3	[0,0], [3,0]	0xc80 000 | 0xc83 000
4	[0,1], [0,1]	0xc80 004
5	[0,1], [1,1]	0xc80 004 | 0xc81 004
...	...	...

Table 1.4: Steps covered for each data entry.

Thus, the TLB experiences a high-frequenct miss (the queried translation not in TLB cache), as shown in Table 1.5.

Step	TLB1	TLB2	TLB3	Miss
0	0xc80 000	/	/	1
1	0xc80 000	0xc81 000	/	1
2	0xc80 000	0xc81 000	0xc82 000	1
3	0xc80 000	0xc83 000	0xc82 000	1
4	0xc80 000	0xc83 000	0xc82 000	0
5	0xc80 000	0xc83 000	0xc81 000	1
6	0xc80 000	0xc82 000	0xc81 000	1
7	0xc80 000	0xc82 000	0xc83 000	1
8	0xc80 000	0xc82 000	0xc83 000	0
9	0xc80 000	0xc81 000	0xc83 000	1
...	...	...	...	

Table 1.5: Steps covered for TLB miss.

Thus, during the whole iteration process, we have $1023 \cdot 3 + 4 = 3073$ times of miss. This is a very low-efficient implementation of code but most programmers ignore it. Essentially, modern compilers have the functionality to adjust these double-iterations automatically: they **revert** the inner loop and outer loop. In other words, the code snippet now becomes as below:

```
int x, y;
result = 0;
for (y = 0; y < 4; y++) {
  for (x = 0; x < 1024; x++) {
    result += matrix[0][x] * matrix[y][x];
  }
}
```

With this method, apparently only 4 times of misses will happen, one for each `y` value. What a time-saver! Here it's important to notice the difference between paging and **segmentation**. Paging is managed by the operating system (which means it's invisible to user applications), and it is a memory management technique used for virtualization and indirection. On the other hand, segmentation is visible to user applications and manageable by the user (user application programmer). For example, if the user declares an array using `a[10]`, then the base address of `a` will be determined by the operating system and the offset will be 10. Thus, if the user tries to access a memory address outside the limit the operating system throws a `Segmentation Fault`.

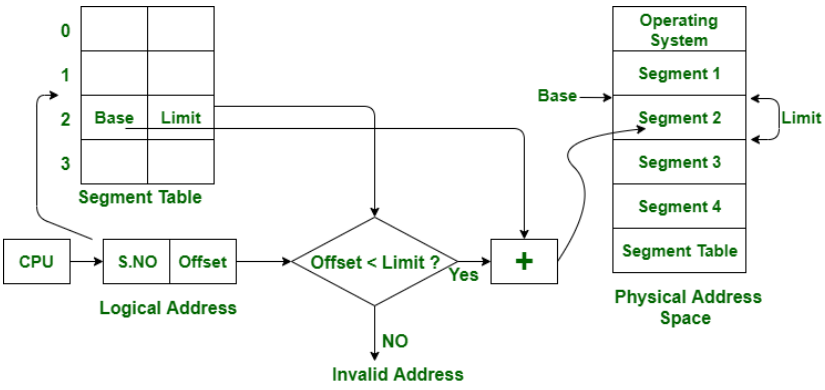

Figure 1.15: The illustration of the segmentation system.

1.4.5 Unfair Paging

The unfair paging scheme is a special approach of paging that translate a virtual address into a physical address with a different address length. Designing such MMU schemes is challenging but useful for learning why the modern operating systems use fair paging scheme. We'll take an example below - Windows' PAE (Physical Address Extension). Say I'm now translating a 32-bit virtual address (4GB) into a 64-bit physical address (much, much larger...)[17]. The page directory takes 128 bytes ($128 * 8$ bits) in memory, each 64 bits long, so it has 16 entries, which means that the virtual address has 4 ($2^4 = 16$) bits for it. As the physical pages are 4KB each, we have 12 ($2^12 = 4K$) bits for it. Now if we have 2 page tables hierarchically and with the same bits in virtual address, we should have $\frac{32-4-12}{2} = 8$ bits for each page table, as summarized below in Table 1.6.

Struct	Offset Length	Entry Length	Entry Amount	Size
Page Dir	4	64	16	128B
Page Table 1	8	64	256	2KB
Page Table 2	8	64	256	2KB
Page	12	8	4K	4KB

Table 1.6: Summary of the Windows PAE Paging Scheme.

[17]This implies that the mapping is **incomplete** because we can never represent all the physical memory after paging.

The design here means that the OS now has more memory to spread stuff out over - there will be less sharing between processes and hence some possible performance benefits.[18] In the traditional paging system, when each program has 4GB virtual space but only a small amount in it is used in the physical memory, it's relatively fine. But when each program takes approximately 4GB of physical memory, the operating system has to page out some of the run-time information of other concurrently running programs from physical memory to the file system temporarily for use of the currently running program, thus causing an unexpected high overhead for memory swapping. However, as an additional level of translation is needed (we have PT2), this scheme should not be effective under circumstances where we've got no program with too large memory usage.

1.5 Memory Allocation

Memory allocation is another important topic in Linux. The so-called `allocation`, means inside the user space or kernel space, we designate one particular part of memory space for preserved use. The allocation system aims differently from the paging system, because the paging system aims to create an abstraction of memory for the user use, while the allocation system aims to preserve a bunch of memory that is **already paged** for the program or **will be paged** separately (for the free page allocator). As an example for the already paged memory allocation, if the user wants some memory in the user space to be reserved for a struct instance on the user heap, he has to use the `malloc` function. This can only be achieved when the user has already assigned a page for its use, so that he can use the heap **inside the page**. In other words, if the user program cannot get his own page, it's impossible for memory allocation.

1.5.1 Allocation Types

There are four main types of memory allocation in kernel mode. The most used one is **kmalloc**, which is used when only a few small items are needed to be allocated. This occupies most of the case because both users and kernels often allocates for a simple pointer or small struct. The second type is the **slab allocator**. When we require lots of repeated items, we use the slab allocator, which caches the deleted items. For example, when a structure is allocated and deallocated for several times the slab allocator detects this behavior, thus next time when allocating memory for this struct, Linux automatically changes the `kmalloc` allocator to the `slab`

[18] `https://serverfault.com/questions/3342/how-does-a-32-bit-machine-support-more-than-4-gb-of`

allocator. The third type is the **free pages allocator** or page allocator, which is used for allocating a big, **physical** contiguous memory region. For most of large chunks of memory requests, we don't need them to be physical contiguous. Thus, we choose to leave the precious resources for functionalities with strict performance requirement, and thus use the vmalloc allocator as shown below. When we need a large area in the virtual memory instead of contiguous physical memory, we use the **vmalloc** allocator. This allocator is especially useful when we want to a large chunk of memory, but they're not necessarily physically contiguous, and thus called **virtual memory allocation**. Using the `vmalloc` allocator, we can get virtual pages that maps to sporadic pages in the physical memory.

1.5.2 The Buddy System

The buddy system lies under all the allocation types mentioned above. For example, after the `kmalloc` allocation is invoked, the buddy system takes over the allocation and see what it can do to fulfill the functionality. If it fails to allocate `kmalloc`, it returns an error code and `kmalloc` will also error out and show the user. Memory is allocated in the smallest unit of pages. After allocation and deallocation, there will be more and more external fragmentation in the memory, which leads to failures of subsequent allocations. The **buddy system** is invented to control the place of allocation to avoid this issue, by allocating the **buddy** that is near the allocated memory first. The buddy system is essentially a bit-mapped binary tree that locates in a dedicated place in the memory, as shown in Figure 1.16. Let's take each pages as 4KB for example, thus the figure shows that this memory area has 32KB in total. As we see, the binary-tree bitmap on the right

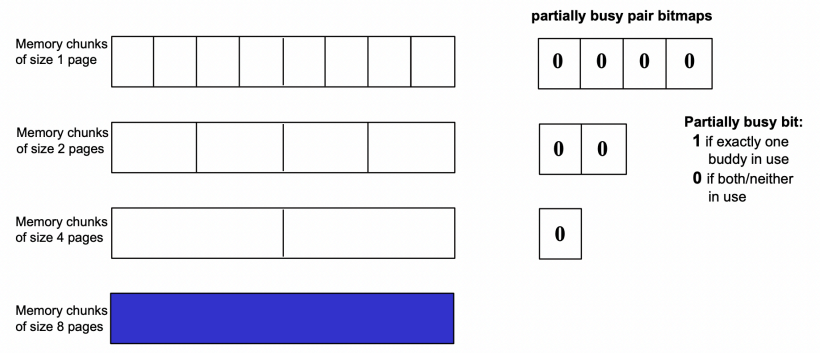

Figure 1.16: The initial state of the buddy system.

is a bitmap, with each bit representing the status of the pair of buddies in the memory. For example, on the first row, we see there are 8 buddies in total, so the number of bits in the bit map will be 4. Bit 0 is in charge of the first 2 buddies, bit 1 for second 2 buddies, and so on. When the allocation status of the two buddies are not the same, the state of this bit becomes **betrayed** and thus the bit should be mapped to 1. Here we go through a working example to show what happens when we allocate pieces of memory. At the very initial state, all the pages are free, so all the bits are 0. If this is the case, the situation is either all the pages are free or all the pages are busy, and we distinguish these two states using the bit from an even higher tree node, which is not shown in the figure above. Now we want to allocate 1 page, so the buddy system must be able to allocate 1 or more than 1 pages for me. The buddy system firstly take a look at the highest layer. After finding that it's all 0, it means that this memory is **not accessible** - no matter it's both allocated or it's both free, it's not accessible. The buddy system then looks down layer by layer, but finds that all of the layers are not accessible. Thus, the memory is divided into two 4 pages of memory, and the memory with 8 pages is **gone**, which means that if there is a request for 8 pages to the buddy system, the buddy system returns error. This process recursively goes to the second layer, and up to the first layer. After splitting the first layer, the buddy system finds a buddy with only 1 page, so this is the buddy to be allocated. The new state become `0001`, as shown in Figure 1.17. Now if we want to allocate another page,

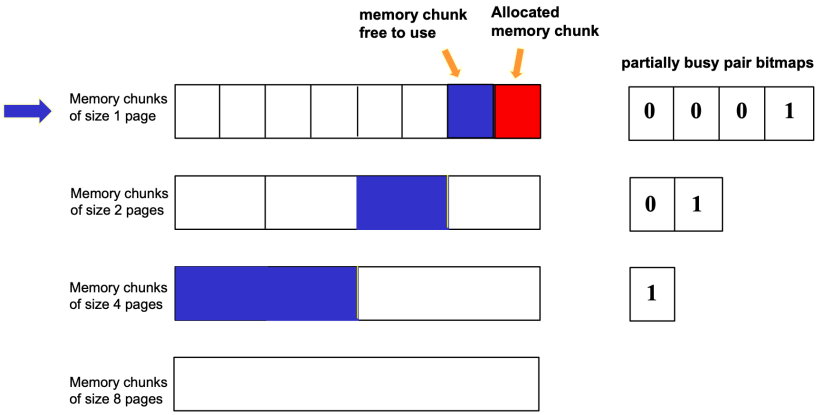

Figure 1.17: The state of the buddy system after allocating a single page.

the buddy system looks into the first layer again, because the first layer contains buddies of 1 page, looking directly into layer 2 wastes resource. After finding that

the last bit is 1, it decides that there is 1 free (accessible) page in the last pair of buddies in the first layer. Thus, it allocates one pages and make the last bit of the first layer to be 0 (not accessible). This rule can be applied to deallocation as well. Now say we want to deallocate the first page we allocated. The buddy system does not need any more information other than the bitmap on the right. After looking at the bitmap of the first layer, it decides that it the last bit (not accessible) must be **both occupied**, because we're deallocating one buddy in this pair but the bit is 0, meaning both are allocated or free. Because we want to deallocate, they must both be allocated. Thus, this bit again becomes 1 and every bit on the lower layers remain the same.

1.5.3 Memory Maps

When we load a program into the memory, we need to link the program with many other dynamic libraries, like `libc.so` and `ld.so`. After loading the dynamic libraries, the image of the libraries will be stored into the user-space memory. When two programs share the same libraries, they use **memory maps** to share memory and avoid repeated loading. This trick is done by loading the libraries into the physical space only once but call the `mmap` function to map this physical address to another virtual address, so that the two programs can use the same shared libraries. An example is shown in Figure 1.18. `mmap` is typically used for

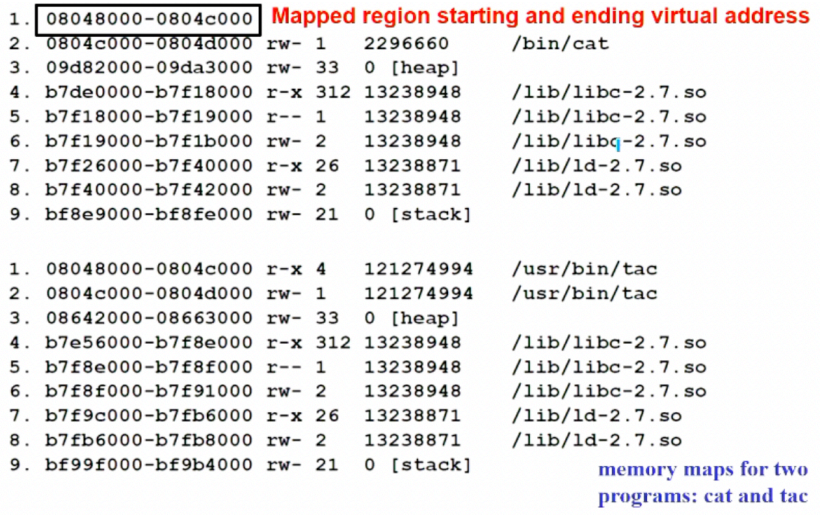

Figure 1.18: An example of `mmap` for the user programs `cat` and êxttttac.

loading a file into the memory. After `mmap`, the file in the disk is **directly** mapped to the memory, so any changes to the file on the disk maps to the memory at once, and any program changes maps to the disk as well.

1.6 Processes

The process is the smallest unit for scheduling, and also a name of data structure called `task_struct`. This `task_struct` is exactly the same as a process descriptor `pd` or a process control block (PCB) as a terminology. Thus, the PCB is also a C `struct` containing the ID, condition, virtual address space, etc. of the process.

1.6.1 Kernel Stack

As discussed in the memory part, we have only one kernel space memory, which is 4MB large. However, now we see that multiple processes can exist at the same time, thus they share the same kernel memory. From this point, we clearly see that the kernel cannot contain massive data for each process, and those should be stored in the user memory space. There is a question about whether the kernel stack can be omitted, leaving only the user stack for each process. The answer is absolutely **no** because we need the kernel to get access to the functionalities provided by the kernel, like system call. Without the kernel stack, a user program becomes almost useless. Here we discuss how the kernel stack looks like. As we have 4MB for the kernel, and each program should have a kernel stack storing per-program data, the size of the kernel stack should be small. Different OS's have different implementations on this selection, but ECE391 in UIUC makes the kernel stack at most 8 KB, as shown in Figure 1.19. We clearly see that each task obtains a kernel stack moving up from the bottom of a kernel memory section of 8 KB. On the top of the section, there is a **thread info** struct which keep a pointer to PCB, which will be discussed in Section 1.6.3 in detail. As we see, this implementation does **not** allow recursion in kernel, otherwise the kernel stack goes up too fast and thus overflowing the section and crash the whole kernel. In other words, because recursion is still considered as one process (with a lots of function calls), the kernel stack of the process will expand beyond the limit very quickly.

1.6.2 Task State Segment

The Task State Segment (TSS) is an important structure in Linux for task scheduling. When a process changes its privilege (go between kernel and user space), or

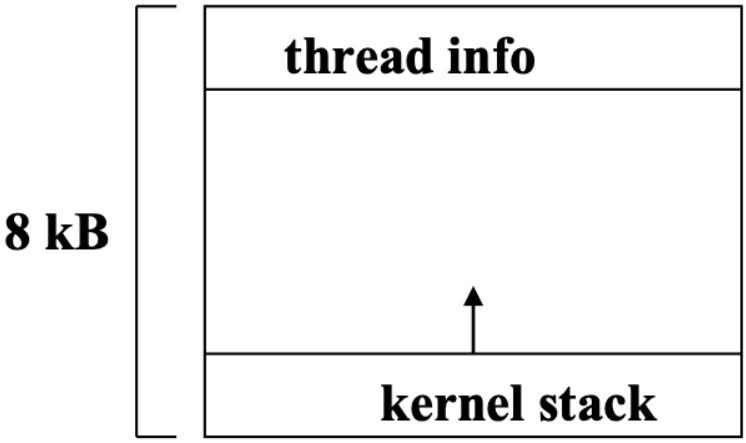

Figure 1.19: An example implementation of the kernel stack for each task.

multiple threads are running concurrently, we need to switch the contexts. However, the operating system has no idea where the stack is. In other words, the operating system needs to be notified the context registers, especially the stack segment descriptor (`%ss0`) and the pointer to the stack top (`%esp0`). TSS is created for this purpose. There is only 1 TSS in the CPU, which means that multiple tasks use the same TSS for temporarily storing their register information. When experiencing a context switch from kernel to user, TSS should update its register information using the kernel stack information. Using pseudo-code, the content update in TSS during in-task context switch can be represented below, where `cpu.esp` represents the ESP register in the running environment, `tss.esp` represents the user stack pointer of this process, and `tss.esp0` represents the kernel stack pointer of this process.

```
1 /* switch from kernel stack to user stack */
2 tss.esp0 = cpu.esp // save current kernel ESP
3 cpu.esp = tss.esp // switch to user stack
4 /* switch from user stack to kernel stack */
5 tss.esp = cpu.esp // save current user ESP
6 cpu.esp = tss.esp0 // switch to kernel stack
```

Just like storing the address of GDT into `gdtr`, storing the address of LDT into `ldtr`, the operating system stores the address of TSS into `tr`. Except for the TSS that `tr` points to, there are many other TSS, but not functioning. In other

words, there is always only one TSS in use, but there are multiple TSS in the memory. When there is only 1 CPU but multiple processes, the CPU is executing the processes concurrently. This is not parallel because the CPU cannot execute both processes at the same time. Instead, it uses a time-sharing strategy for all of the processes fairly, as shown in Figure 1.20. The time-sharing strategy needs each process to have a "memory" of itself when it ends so that it can retrieve whatever it had next time when it's active. Thus, we need a separate TSS for each process. For example, in Figure 1.20, we need 3 TSS, for process 1, 2, 3 separately. When the time slot allocated for process 1 ends, it writes its current state into the TSS for Process 1. Then the CPU moves on to process 2. After process 2 ends, it stores the current state into another TSS and the same for process 3. After process 3 is stored in the third TSS, we come back to process 1, so before it starts we load the information in TSS for process 1. When the time slot for process 1 is used up, we store the current state in TSS for process 1 again and load the information in TSS for process 2 to process 2 run time environment, thus forming a loop.

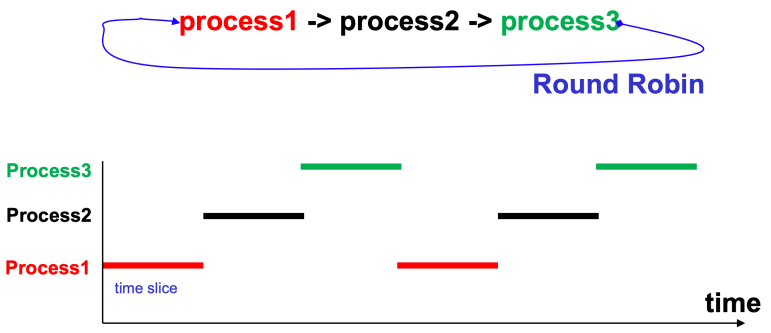

Figure 1.20: An example of time-sharing strategy for 1 CPU with multiple processes running.

1.6.3 Process Control Block

The **Process Control Block** is the structure to describe one process (task), so it's also called a **Task Structure**. It contains everything needed to describe a task, including the PID (process ID) of the task, the parent PID of the task, the file descriptor table of the task, the command and arguments in command line that launches the task, and helper fields like values in all kinds of registers, especially `esp0` (kernel stack pointer) and `esp` (user stack pointer). Before switching tasks,

the values of `%ss0` and `%esp0` in TSS should be stored in PCB, so that when the time slice comes back to this process, it can refill TSS with the stored information. As mentioned, there is only one TSS in the CPU, so it's impossible to store information of multiple processes inside the only TSS structure, which explains why we need to store `tss.esp` and `tss.esp0` inside the PCB before we want to do the task switch. Explaining this rule with code, we have

```
/* before switching from p1 to p2 */
pcb1.esp = tss.esp // store kernel stack ESP
pcb1.esp0 = tss.esp0 // store user stack ESP
/* after switched to p2 */
tss.esp = pcb2.esp // restore ESP for p2
tss.esp0 = pcb2.esp0 // restore ESP for p2
/* before switching from p2 to p1 */
pcb2.esp = tss.esp // same as above
pcb2.esp0 = tss.esp0
/* after switched to p1 */
tss.esp = pcb1.esp
tss.esp0 = pcb.esp0
```

Hitherto, we've discussed two cases when we need to update TSS. Conclusively, when we do an in-task context switch or we switch to a different task, we need to update TSS. Besides, we only need to store TSS (into PCB) when we switch to a different task, because there is only 1 TSS and we want to reserve its value of the process that the program is going to leave.

1.6.4 Per-task File Descriptor Array

As mentioned in Section 1.6.3, each task stores a file descriptor array, which contains up to 8 files. This is called **Per-task File Descriptor Array** or **File Descriptor Array**, because it stores the opened files inside a task. When a task is launched, we have 2 default files opened: the `stdin` file and `stdout` file. Strictly speaking, these two objects are not files, because they're not opened from the file system - rather, they're **fake terminals**, because

1.6.5 Process Tree

All processes except `init` (or `systemd`) must be a child process of another process because we must have one process to call this process so that it can be executed. The creation of the process is called `fork()`. All the processes are stored in the

structure called `task_struct`, and the linked list is shown in Figure 1.21. You

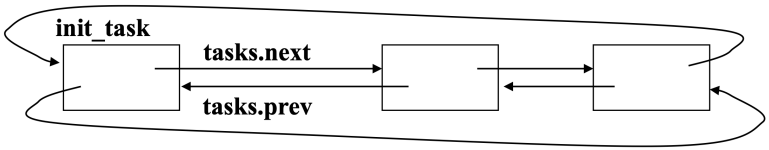

Figure 1.21: The structure `tasks` in the kernel space with prototype `text_struct`.

can use `pstree` to list all the processes in a tree view. The terminal output is shown in Figure 1.22. We can use `fork()` to create a new process. Note that the

```
release/mp7/printRev$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─agetty
        ├─at-spi-bus-laun─┬─dbus-daemon
        │                 └─3*[{at-spi-bus-laun}]
        ├─at-spi2-registr───2*[{at-spi2-registr}]
        ├─avahi-daemon───avahi-daemon
        ├─boltd───2*[{boltd}]
        ├─colord───2*[{colord}]
        ├─cron
```

Figure 1.22: The terminal output of the `pstree` command.

created child process is a **clone** of the parent process after the `fork()` statement. For example, in the code below, we have the child process and the parent process executing the `printf()` statement both. Note that after experiments, it's not decidable which process (parent or child) will execute the `printf()` process first, but there will always be two `Hello World!` appearing in the terminal.

```
1 fork();
2 printf("Hello␣World!\n");
```

Also note that Linux uses a **share-on-read, copy-on-write** forking rule to minimize data copy. In this rule, only when the data need to be written, we copy the data from parent process to child process (deep-copy). But when the data is not

written in the parent or child process, it uses the data stored in the parent process (shallow-copy).

1.6.6 Process Lifecycle

Each process has its lifecycle after creation in Linux, as shown in Figure 1.23. After the `fork()` handler in the kernel is executed, a process is created and begins its lifecycle as a Finite State Machine (FSM). When the process is in the

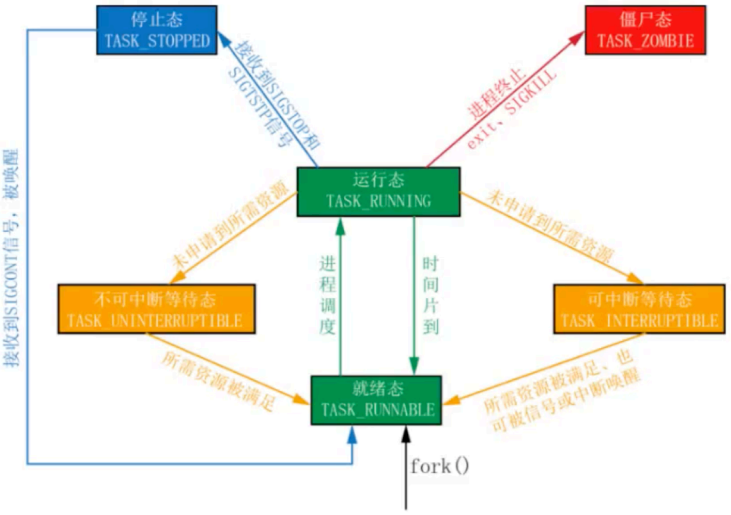

Figure 1.23: The lifecycle of a process inside Linux.

`TASK_RUNNABLE` state, it will be assigned to the system priority queue for waiting for execution. A process with higher priority will be assigned to the fronter part of the queue. After the system task scheduler detect that the process is the next one to execute, this process will be assigned a **timeslice** (or quantum/processor slice) and get to the `TASK_RUNNING` state. A timeslice is the period of time of assigned to the processor for execution before **preempted** by other processes by **time-sharing operating systems**. After the timeslice for the executing process is used up, the process will come back to the `TASK_RUNNABLE` state. When the process is in the `TASK_RUNNING` state, and has completed all its tasks, while it had not been cleaned by its parent process, the state becomees `TASK_ZOMBIE`. In another way, when the parent process is killed but the child process is still there, it will become an orphan process and will become the child of the `init` process. To

examine all the process running, it's preferable to use the command `ps -aux` or `ps -ef`. After getting the pid (process id) of the process, you can kill the process by the command `kill <pid>`.

1.6.7 Inter-process Communication

Inter-process communication, or process synchronization, is a mechanism to guarantee the correct workflow of the whole program when multiple processes are running concurrently or in parallel. The basic synchronizations include spinlocks (busy-waiting), semaphores (idle-waiting), and conditional variables (conditional waiting). All synchronization strategies are essentially blocking strategies. The **blocking strategy** is the behavior of a process to stop executing the code until a signal is received from another process. For example, if some code in one process have to be executed after some resources have been released by all other processes, this process has to wait before the code is executed, thus it experiences a blocking action.

1.6.8 Daemon And Background Processes

A special kind of process in Linux is called the daemon process. This kind of process will be launched every time when the system boots. This kind of process is always an orphan process, as it does not bound to any parent processes (except the `init` process). Daemon processes usually ends with `d`, like `systemd` for system daemon processes and `httpd` for Apache HTTP service. The creation of a daemon process takes 2 steps. Firstly, we need to make the process a background process using `nohup &`, as described below. Secondly, we need to detach the process from its TTY.[19] At last, we need to make the process into service and launch every time when booting using `systemctl`. Another kind of special process is called the **background process**, which does not launch at the boot time, but still cannot be terminated after the terminal is closed. After using the background process, it will run until an explicit `kill` command is executed. Also, after the background process is launched, all information produced by the daemon process won't show in the terminal. Any processes except the daemon and background processes will be killed after the terminal is killed because the terminal is the parent process to all processes it creates (**spawns**). In this case, we can take the background/daemon process as a special case of the orphan process. Note that a background process can't be launched by adding `&` at the end of the command. The processes created

[19] https://unix.stackexchange.com/questions/266565/daemonize-a-process-in-shell

by appending a `&` are still appendix to the terminal and will be terminated after the terminal is killed. To launch a real background process, use the `nohup` command with the syntax `nohup <cmd> &`, so that after the terminal ends the process is still active, due to the fact that it's now an orphan process. Due to the fact that the background process and daemon process do not bound to any shell. we can't use the standard in and out `stdin` and `stdout` for the process (because no shell is in charge of it). In this case, the log files are mostly done by redirecting the output to files. The difference of a normal process, an unblocking process, a background process, and a daemon process is shown below.[20]

Type	BootLaunch	PPID	TTY	Approach
Normal	No	Caller	Shell	Call
Unblocking	No	Caller	Shell	Postfix `&`
Background	No	1	Shell	Prefix `nohup`, Postfix `&`
Daemon	Yes	1	None	`nohup,&,systemctl`

Table 1.7: Comparison of a normal process, an unblocking process, a background process and a daemon process.

1.7 Threads

As Linux processes has lots of complex scheduling strategies and large overhead for creating and killing a process, in software computing area, it's not recommended to create processes for parallelism. Rather, software engineers tend to use multi-threads to program in parallel.

1.7.1 Lightweight Process

However, in Linux, a thread is essentially still a process, because Linux does not provide a different implementation for a thread. A thread, instead, is a lightweight process that always uses shallow-copy when forking. A thread can access all the resources in the process that it belongs to, and thus all threads in the same process have access to all the resources in the process. The relationship between thread and process can be expressed like that, we've got a `pid` (process id) for each process, while we can get a `tgid` (thread group id) for each thread, because a process is independent while a thread is often formed into groups, and the whole

[20]In Linux, a TTY of None is marked with "?".

bunch of threads forms a process (which is different from the implementation of one process with only one thread). A graphical illustration of the principles of the threads is shown in Figure 1.24. Note that a thread can have its own PCB, just like a process. Because thread is not itself a command executable by Linux (it's

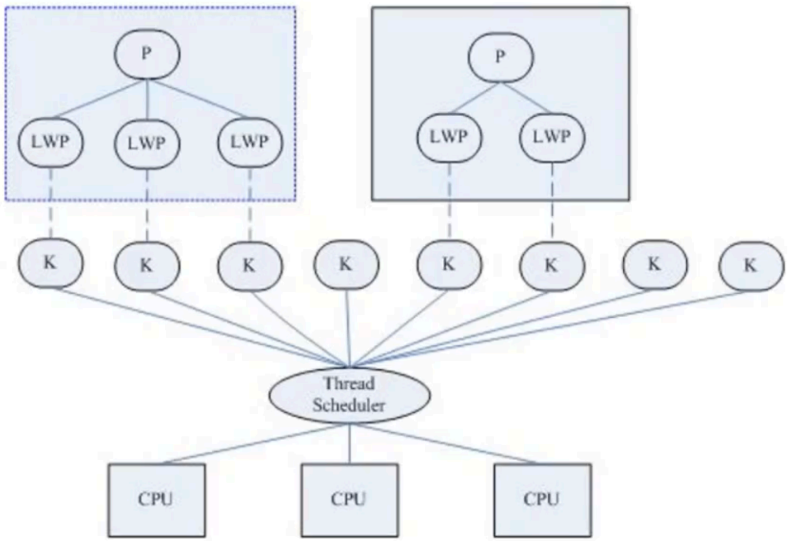

Figure 1.24: Threads in Linux. **P**: process. **LWP**: Light Weight Process. **K**: task id.

a process), it has to be encapsulated inside a library. Thus, the implementation is left for different programming languages. The official programming language for threads is C, and the typical programming languages supporting threading is shown in Section **??**.

1.7.2 User-level Thread

The user-level threads are implemented by users and the kernel is not aware of the existence of these threads.[21] As shown in Figure 1.25, the user-level threads run in the process in the user space, and is scheduled by a thread table inside each process. The user-level thread is controlled by a TCB (Thread Control Block) instead of a PCB. In this case, switching from one thread to another does not require a context switching between the real mode and protected mode, which makes the thread much more light-weighted. However, an important drawback

[21] https://www.tutorialspoint.com/user-level-threads-and-kernel-level-threads

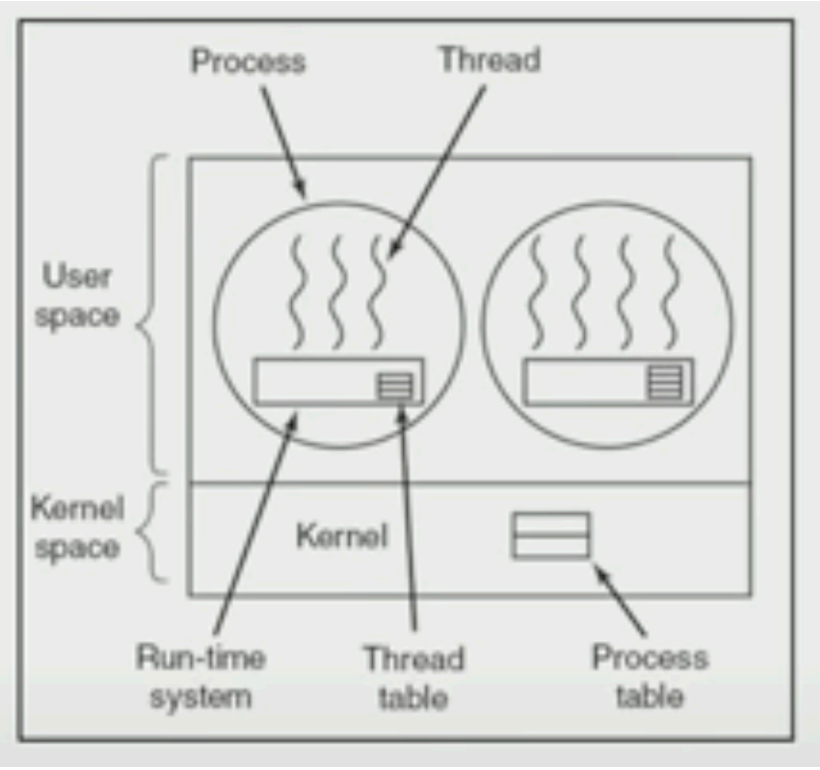

Figure 1.25: The system of user-level threads.

also appears that, because the kernel (OS) is not aware of whether the process is multi-threaded, it is not able to judge how much timeslice it need to give for each of the processes. For example, process 1 has 100 threads but process 2 has 1 thread, but the OS will assign them the same amount of timeslice, which causes a time waste due to blocking issues.

1.7.3 Kernel-level Thread

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. For the kernel-level threads, multiple threads of the same process can be scheduled on different processors in kernel-level threads.[22] As shown in Figure 1.26, the TCB is in the kernel, as same as PCB. This mecha-

[22]https://www.youtube.com/watch?v=LNiNOW-_8lw

nism enables the kernel to see how many threads there are in a process. Therefore, the OS is able to decide how much timeslice and how many processor cores it need to allocate to each process and thread. However, this approch is also slow because the thread table is not in the same space as the threads, so it takes lots of overhead to perform such a scheduling.

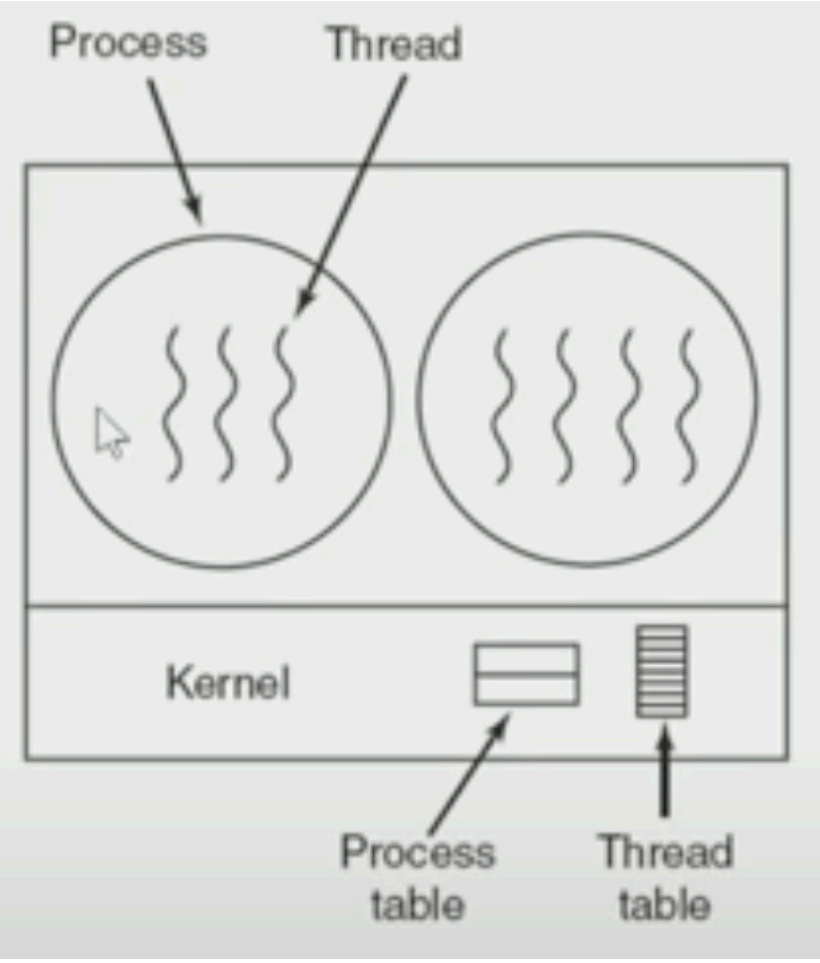

Figure 1.26: The system of kernel-level threads.

A comparison of user-level thread and kernel-level thread is shown in Figure 1.27.[23]

[23]http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

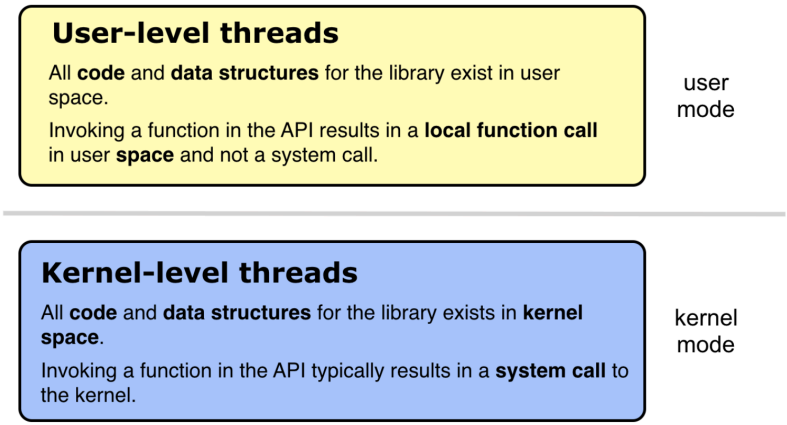

Figure 1.27: The conversion of user-level thread and user-level thread.

1.7.4 Models of Mapping

The mapping from the user-level thread and the kernel-level thread has 3 basic types: many-to-one, one-to-one, and many-to-many. In the many-to-one mapping, lots of user-level threads are created, but only one system call will be raised to invoke a kernel-level thread call. Thus, the program is essentially running on one kernel-level thread. In the one-to-one mapping, one user-level thread is implemented to correspond to one kernel-level thread, so creating multiple user-level threads is also spawning multiple kernel-level threads. In the many-to-many mapping, m user-level threads is mapped to n kernel-level threads, and the allocation is done by a separate scheduler. As mentioned by wiki[24], the `pthread` library in C is a one-to-one mapping: "NPTL is a so-called 1*1 threads library, in that threads created by the user (via the pthread_create() library function) are in 1-1 correspondence with schedulable entities in the kernel (tasks, in the Linux case)."

1.7.5 Hyper-threading

The hyper-threading appears within the most recent 10 years. As it's not always possible to make the number of cores inside each CPU larger, Intel smelled such a way that can increase the number of "cores" inside 1 core. Note that hyper-threading is **not** adding more cores inside the system, but it schedules different

[24]https://stackoverflow.com/questions/8639150/is-pthread-library-actually-a-user-thread-solution

tasks to keep each core always busy. In this way, it shouldn't be counted as a valid core-expansion strategy.

In the hyper-threads, each core has more than one register set, but no additional execution units, and the hyper-threads are scheduled more or less evenly[25].

Technically, HT (hyper-threading) should be called SMT (simultaneous multi-threading). In 2004, Intel announced its first usage of hyper-threading on its Pentium IV processor.

1.8 Concurrency And Scheduling

In this section, we discussed about the scheduling system in the Linux system. The notion of **scheduling** was put forward because there may be multiple tasks running at the same time, but not all the processors are available at that moment. For example, when there are three programs running but only 2 processors available, one task will be left out if no strategies are applied. Only after the first two tasks are done, the third one is executed.

This implementation is extremely bad, because there are different types of tasks that need different priorities. For example, when a file I/O task is enqueued, it needs to be executed immediately so that the task that launches this task can go on its process. Thus, we need to ensure that whenever a task comes, it need to be served due to a strategy that is **fair**.

The idea is to use a strategy called the **time-sharing** strategy. The processor is now executing not a task, but a **time slice** for the task. In other words, the smallest unit for task excution is now not a task, but a time slice for some task. This policy is called **concurrency**, because when the switching is fast, two task seems to be running on a single CPU at the same time, and this strategy is called **Round-Robin Strategy**, because it's like holding a round competition. The length of each task should not be too long or too short, as a too long task leads to similar cases as the original one, and too short task makes context switching a time consumer.

When a time slice comes, the scheduler decides which task should be executed on which processor using the strategy specified. Then this process has the time slice, which means that it can run in this time slice. Note that when an interrupt comes in the time slice, it takes the time of the process. In other words, the interrupt consumes the process's time, instead of extending it.

[25] https://stackoverflow.com/questions/18831996/hyperthreading-code-example

1.8.1 Scheduling Routine

The process scheduling is done using a complex system, but we only explain it in a high-level way, as illustrated in Figure 1.28.

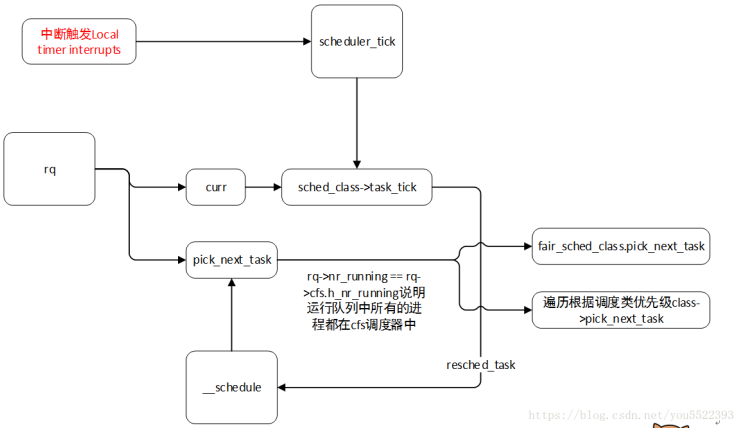

Figure 1.28: The flow graph showing how the scheduler is called (triggered) and how it controls the workflow of the processes.

There is a device called Programmable Interval Timer (PIT), which generates interrupts to IRQ0 (highest priority), connected to PIC. This timer has a fixed interrupt rate so nothing can change its frequency, which is different from the Real Time Clock. Every 1000Hz, it sends out a signal to PIC, which is handled so that the scheduler is called.

The scheduler function is called `scheduler_tick` in the Linux kernel code. This function calls `task_running_tick`, which forwards the running task by a **tick** (time slice). The function then calls the actual scheduling subroutine `__schedule`, which decides what strategy to use and thus how to arrange the current task and next task.

After this function is done, it calls the `task_switch` (or some literature calls it `pick_next_task`), which moves on to another task if the next task is not the current task per se.

When doing scheduling, a context switch needs to be conducted as well. The process is pretty similar to the rules described in Section 1.6.3. Say the three tasks are `python, shell, cat` and we use the round-robin scheduling strategy. Then every time before we switch to `shell` from task `python`, we need to store the

`python` TSS information in the `python` PCB. Then at the very beginning of the task `shell`, we need to restore the `shell` TSS with those stored in `shell` PCB.

Note that the context switch process should be in the scheduler, before the actual scheduling process begins. As each time the scheduler is called when PIT sends an interrupt, the context switch should happen for every PIT interrupt, which makes sense.

Thus, in `python` PCB, we store the `python` user context, PIT handler (handling a PIT interrupt), and the scheduler. A simplified process of a scheduler is shown in Figure 1.29.

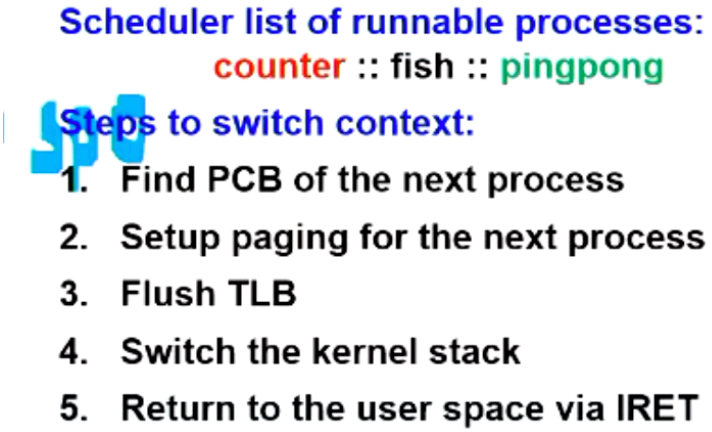

Figure 1.29: A simplified version of scheduling, which is used in MP3 in ECE391, UIUC.

1.8.2 Important Notes of A Scheduler

Assume the same condition as mentioned in the last section. If the scheduler is absent, only one process can run at the same time in a uni-processor system. Thus, only after the first process `python` ends, the second task `shell` will be run. In other words, scheduler enables the operating system for multitasking.

Without the scheduler, the number of running processes is always the same as the number of processors, because the processor is always occupied and blocked by this process. The scheduler is designed to switch among different processes and thus allow multiple programs running at the same time. Without the scheduler, in a uni-processor system, a video cannot play when the timer is on, which makes the computer extremely incapacitated.

Another question is that, assume we have two counter programs running, staring at the same time and ending at the same time. However, one counter displays the number onto the screen while the other not. The questions is, do the counters end with the same number? The answer is no, because displaying numbers onto the screen takes time. When scheduler gives every process with the same amount of time, not same amount of cycles.

1.9 Device Drivers

In this section we discuss about the definition, implementation, and example of the device derivers. A **device driver** is a kernel-space software that can get access to the hardware, and can also offer functionalities to the user program. Device control operations are defined/implemented by a device driver specific to a given hardware device. Using a device driver, the programming interfaces of the device becomes well-defined and out-of-box. Thus, the device driver is essentially an abstraction of the physical device.

A device file is usually **stored** as a real file, thus it can be opened by the program in a normal way. Inode includes an identifier of the hardware device corresponding to the character or block device file. Thus, when loading the device file into the program memory, it automatically maps to the hardware device, including the read, write, open, and close functions implemented in the device deriver. Each system call issued on a device file is translated by the kernel into an invocation of a function of a corresponding device driver.

The approach to determine the version of the device (or what model it is) is to use the major number and minor number of the device specified in the `/dev` directory of the Linux system. The **major number** is used to specify the device driver of the device. In other words, if two devices have the same major number, then they are the same type of device and thus use the same device driver. To view the major numbers of the loaded devices in Linux, type `cat /proc/devices`. The hard disks usually have major number 3. When two devices have the same major number, they use the **minor number** to determine different devices. When two devices have the same major number, they have to obtain different minor numbers (usually consecutive). For block devices, the minor number also distinguishes the sector of the block device, because each sector is individual in the `/dev` directory, as shown in Figure 1.30.

There are some other devices shown in the `/dev` directory except for the character devices and block devices, including the FIFO pipeline (`|` shown in the linux CLI commands), sockets, hard and soft links (like stdin, stdout and stderr), and

```
brw-r-----  1 root   operator   0x1000001 May 10 09:09 disk0s1
brw-r-----  1 root   operator   0x1000002 May 10 09:09 disk0s2
brw-r-----  1 root   operator   0x1000003 May 10 09:09 disk0s3
```

Figure 1.30: The output of the command `ll /dev` on MacOS.

directories.

To install or remove a device driver compiled as a kernel module, use the `insmod` or `rmmod` commands.

1.9.1 Implementation

As mentioned, everything (files and devices) in Linux are files. A following question is, how can the operating system tell the difference between a device and a traditional file? The answer is, opening the device with some special operations! For example, the subroutine to communicate with the Real Time Clock (`rtc`) in the user space Linux is shown below.

```
1 int fd = open("dev/rtc", O_RDONLY, 0);
2 int ret = ioctl(fd, RTC_IRQP_SET, rate);
```

The `ioctl` function is a system call used in the user space to redirect the program into the kernel space and call the **device driver**. In other words, in the user space, we communicate with the device by calling the device driver. There is one interesting point in this code: what does `fd` mean? It's a **file descriptor**.

The file descriptor is an integer for the kernel to identify which file/device is being opened and is being referred to. In the given example, `open` returns the file descriptor designated by Linux Kernel, and programmers can use this value for later device operations like `ioctl`. Note that although traditional files can also have a file descriptor, we can't use `ioctl` to send commands to them for manipulating, as they don't have a device driver designed in the kernel.

It's a very natural question, then, about the use of `ioctl`. A typical way for using `ioctl` is using the bunch of code below.

```
1 int ret;
2 ret = ioctl(fd, MYCMD);
3 if (ret == -1) {
4     printf("ioctl:␣%s\n",
5         strerror(errno));
```

```
6 }
```

Firstly, `ioctl` is only callable for a `tty` device, or **TypeWritter** device. The kernel devices can be divided into three types: character device (tty device), block device, and network device. The difference between a tty device and block device is that, a tty device can only be read or write character by character using data streams, while the block device is able to be accessed randomly, like memory and SD card.

The device driver is usually located in the kernel, as it needs higher priority to derive system calls and talk directly to the device controller (which is in the hardware). To install the device driver, you need to build the source code of the device driver into the kernel and load the module. An example of a device driver is the **Tux Driver** mentioned in ECE391 in UIUC, as shown in Figure 1.31.

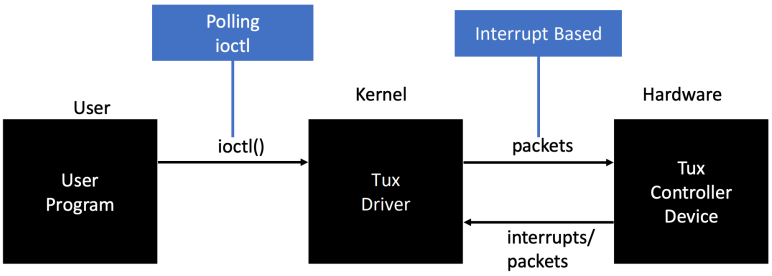

Figure 1.31: The workflow of the Tux Driver.

In the user space, the programmer polls the situation of the device driver using the `ioctl` function call using a `while` loop. The device driver in the kernel, however, does not poll on the hardware, but it sends a protocol-specified packet to the driver and waits for an interrupt with a packet delivered back by the controller device. When it gets the packet, the situation of the driver changes and the user space program detects the change to make further operations.

The device driver of a tty device is also divided into 3 layers: tty core I/O layer (top), line discipline layer (middle), and tty driver layer (bottom), as shown in Figure 1.32.

The figure above mentions about a concept called **faked terminal**. Generally speaking, there are 63 working terminals in Linux at the same time (`/dev/tty1-/dev/tty63`), but there is only 1 focused terminal, which is visible to the user. Except for built-in terminals, Linux also supports **Serial Terminals** using device names

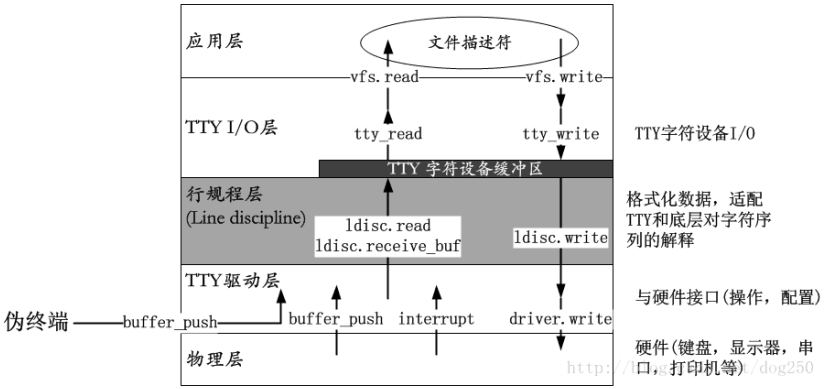

Figure 1.32: Layers of a typical `tty` driver.

`/dev/ttyS0...`, which supports devices using a serial port like VGA and game handlers (including the one we took for example, the Tux Controller). Similarly, the faked terminal just need to implement all mandatory callback functions for a terminal device like redirecting the input by user to the computer and output by the computer to the user. However, instead of a physical terminal, it does not redirect the I/O between user and physical device, but another user-space application and the user. In other words, it provides an interface to other applications, like Telnet and ssh, to connect to the physical terminal.

In the example provided, the Tux controller is essentially a Serial Terminal device. Making it clear, **Tux is terminal per se**. That's the reason why we can use `/dev/ttyS0` to get access to it.

1.9.2 Device Driver Synchronization

Device synchronization is very important but subtle when programming. Consider such a situation that, you've got a game that requires two person to play at the same time, controlling one person. Thus, you need two input devices, say a keyboard and a gamepad, to control the same object. The underlying vulnerability is that, when you're using your keyboard, your friend may be using the gamepad to disorder your signal sent to CPU. Thus, you get mad when playing the game and smash your keyboard onto your friends' head. This should not happen.

To address this issue, firstly, we consider using two threads in CPU for the two drivers to run and listen at the same time. In each thread, we put a `while(1)` iteration to keep the thread running. However, this does not solve the problem

because when you're pressing the buttons at the same time, CPU still gets 2 signals and a race condition happens. The solution is to use a semaphore (or we call it a mutex), as shown in the code snippet below.

```
static pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
void* kb_thread(void* arg){
  while (1) {
    pthread_mutex_lock(&lock);
    // critical section
    if (button_pressed) {
      // do something
    }
    pthread_mutex_unlock(&lock);
  }
}
```

However, this solution is still not perfect - it's a waste of resources. When the button is not pressed, the process is still running in the while loop and thus wasting the CPU computability. Note that the semephore here does not mean idle wait - it has nothing to do with the control flow! The control flow is determined by the while loop, so it's still a busyd wait. Thus, when the button is not pressed, the cycling program will make other programs slow.

The improvement is to use a **conditional variable** to make one thread in the idle wait. After the program encounters the `pthread_cond_wait` command, it goes to sleep (idle wait) and wait for another thread to execute the command `pthread_cond_signal` to wake it up, as shown in the code snippet below.

```
static pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cv =
  PTHREAD_COND_INITIALIZER;
void* kb_thread(void * arg){
  while (1) {
    pthread_mutex_lock(&lock);
    while (!button_pressed) {
      pthread_cond_wait(&cv, &lock);
    }
    // data_available, do something
    button_pressed = 0;
    pthread_mutex_unlock(&lock);
```

```
14    }
15  }
16  // main thread
17  int detect_button_pressed(){
18    pthread_mutex_lock(&lock);
19    button_pressed = 1;
20    pthread_cond_signal(&cv);
21    pthread_mutex_unlock(&lock);
22  }
```

Using this approach, we can avoid busy waiting. There is a trap here that we still use `while(!button_pressed)` here to detect weather the button is pressed, because the thread may experience a **spurious wakeup**, meaning that the thread is waked up at the line it goes to sleep at goes on the whole program without the signal of the conditional variable. This is why we add the while loop for double check. With this additional loop, when the button is not pressed, even if the spurious wakeup happens, it's still in the while loop so it cannot go on to the utility part. On the contrary, when the button is pressed, the conditional variable sends the thread a signal to activate it again and the thread reduces this to 0 again. Because they're atomic operations to each other (which is guaranteed by the mutex), this can operate without glitch.

1.9.3 Devices Drivers With Memory: VGA Driver

There are some more sophisticated device drivers, which have a memory chip in it, like some keyboards and the VGA display. In this section, we focus on the VGA memory and how to manage it, with the explanation of the mapping between memory and display.

Firstly, we need to differentiate from the two kinds of memories: computer memory (video memory, or vmem) and VGA memory. Video memory is shown in Figure 1.33, while VGA is on the VRAM on the VGA device.

The translation of VGA memory and video memory is shown in Figure 1.34. The leftmost graph is the logical view window on the VGA display, and we have 32 pixels in total. The pixels are stored in the "Memory" (video memory) in a normal way: we have 8 pixels in each row, and each pixel takes 1 byte.

As video memory is byte-addressable, we can calculate the memory address by Formula 1.1, where M is the memory address, r is the current row index, $l(r_0)$ is the length of the first row, and c is the current column index. In other words, the memory in the video memory is simply calculated by the flattened position of

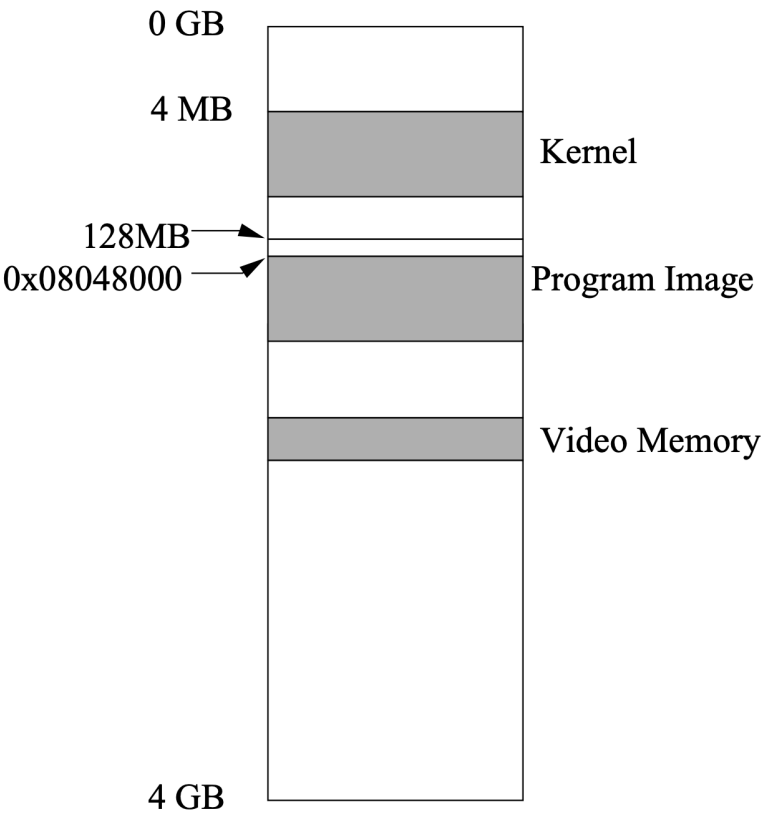

Figure 1.33: The computer memory layout and the location for stroing VGA memory.

the logical memory.

$$M = r \cdot l(r_0) + c \tag{1.1}$$

On the other hand, the VGA memory is not byte-addressable, so not each pixel can be accessed. Rather, the memory is word-addressable, so changing one particular pixel should be managed using a masking approach, that we firstly mask all other pixels in the group of 4 (i.e., in the address), then change the whole address, but because the other pixels are masked and they stay the same, only the byte we want to change is changed.

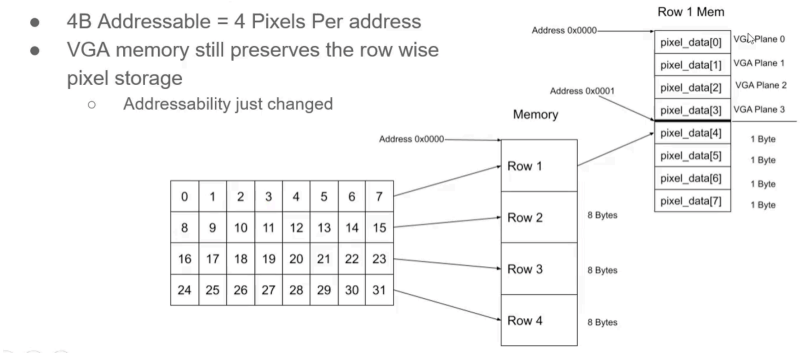

Figure 1.34: The mapping between video memory and VGA memory.

The vmem and VGA memory stores all the pixels in "the world", the **build buffer** stores the shuffled VGA memory in "the world", while the logical window stores the content on the screen. In other words, although the build buffer is itself same size as vmem, only a small part of it is shown on the screen, which is called a **logical window**. The memory location for the logical window in the build buffer is called the logical planes, with an amount of 4, as shown in Figure 1.35.

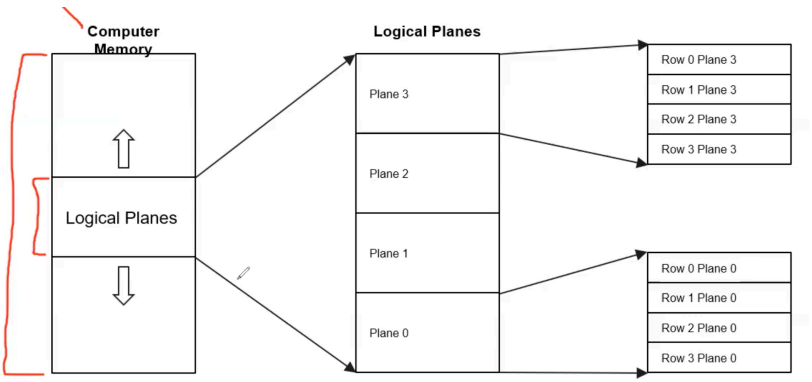

Figure 1.35: The location of logical buffer and logical plane in the build buffer.

The reason why the number of logical planes is 4 is because of the addressability of VGA memory. Because VGA is 4-byte-addressable, pixels should be organized into groups of 4, leading to the number of planes to be 4.

In Mode X of VGA, all pixels on the same column should have the same plane number. Thus, if we move the logical window one pixel down, the logical planes

will be moved one address down as well, as shown in Figure 1.36.

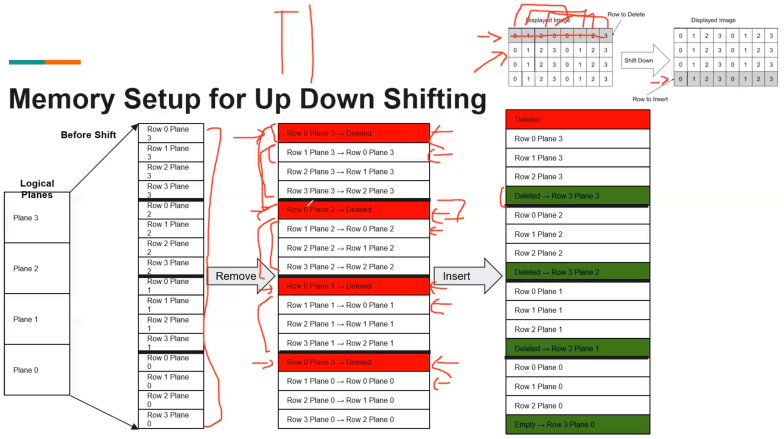

Figure 1.36: The down shift of logical planes in the build buffer.

Similar changes apply to shifting 1 pixel right, as shown in Figure 1.37. Say we now want to move the logical window right by 1 pixel, we see that only plane 0 needs to be modified, as Mode X makes one column representing a single plane. After the shifting, a new column of plane 0 appears. Refering to the memory layout of the logical planes, we see that the first column of plane 0 is deleted, but all other columns of plane 0 are preserved (in this example, we have Px0 deleted and Px1 reserved). Later, we add one column at the rightmost column of plane 0, which is equivalent to adding a pixel following the previous last pixel of plane 0. The final result is shown in Figure 1.37.

Thus, after one right shift, we see the whole plane 0 is shifted down by 1 address (byte), while all other planes does not shift, leading to one empty address between plane 0 and plane 1. If the logical window is shifting right by 1 pixel again, plane 1 in the build buffer will be shifted down by 1 address too, and the empty space will be filled. After 4 logical shifts, the logical planes are now aligned again. Shifting left is just the reverse - you shift the corresponding plane up.

Note the order we store the planes: they're 3 to 0 from top to bottom. If we reverse them, each shift will crash into the plane below them and thus causing hazards.

With the above knowledge, we still need to apply the mapping from logical plane to VGA plane - they're not the same, as shown in Figure 1.38. After shifting right, the logical view tells that a new pixel appears on the rightmost column in

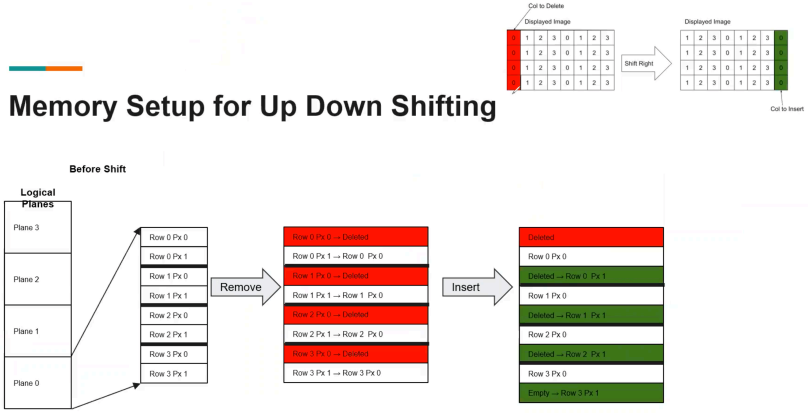

Figure 1.37: The right shift of logical planes in the build buffer.

plane 0. However, from the physical VGA view, pixels 0, 8, 16, 24 are changed to pixels 1, 9, 17, 25, respectively. Thus, the logical view of plane 1 is mapped to VGA view of plane 0.

Similarly, if we removed the leftmost column in logical view plane 1, logical view tells that the rightmost column is now 1, but physical view still gives 0. In this way, we can conclude the mapping as shown in Figure 1.38: 3 mapped to 2, 2 mapped to 1, 1 mapped to 0, 0 mapped to 3.

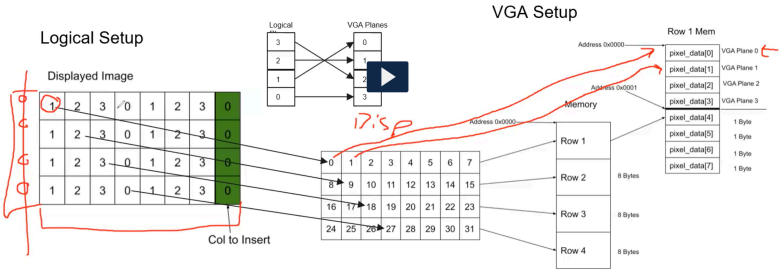

Figure 1.38: The mapping from logical plane to VGA plane.

The formula for calculating the physical shift is shown in Figure 1.39. We need to have `show_x » 2` because we want to get the memory address in the VGA memory due to left/right shift. We have `p_off = 3 - show_x & 3` as the number of planes we need to get down so that we get to the current plane in the logical view. For example, say we now have logical plane 2, which is the second top

in the logical plane, so we need to move 1 plane down (from logical plane 0 to logical plane 1). `p_off < i` is to calculate whether the current address is below the "bubble", which takes 1 memory address.

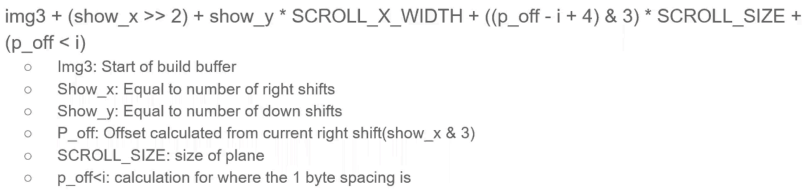

Figure 1.39: The plane shift equation.

The exact code is shown in Figure 1.40.

```
/* Calculate the source address. */
addr = img3 + (show_x >> 2) + show_y * SCROLL_X_WIDTH;

/* Draw to each plane in the video memory. */
for (i = 0; i < 4; i++) {
    SET_WRITE_MASK(1 << (i + 8));
    /* copy content of all planes in sbar_buf into mem_image for display */
    // main is at an offset of video memory
    copy_image_main(addr + ((p_off - i + 4) & 3) * SCROLL_SIZE + (p_off < i), target_img);
    // sbar is at the top of the video memory
    copy_image_sbar(sbar_buf + i * SBAR_PLANE_DIM);
}
```

Figure 1.40: The exact code for Modex showing the screen.

1.9.4 Interactive Device Driver: I-mail

Interactive device drivers are drivers that includes direct user interactions, like `telnet` and `imail`. These drivers need to ensure security because malicious users can do hard to the device driver, or even the whole operating system.

We discuss about the interactive device drivers using a working example: the I-mail device driver. The **I-mail** (Illinois Mail) device driver is a mailbox system that allows sending and receiving messages among several users in Linux.

To support the I-mail driver, we need specific data structures to construct a mailbox. In this design, we construct a hierarchical architecture that starts with the **user list**, while each user list contains a **mailbox** storing all the mails received by the user, as illustrated by Figure 1.41.

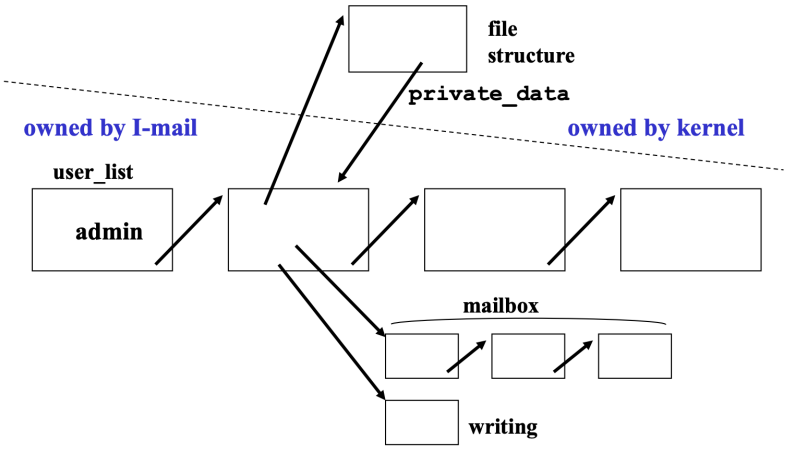

Figure 1.41: The data structures for the I-mail device driver.

In the figure, we can also capture that there is a **writting buffer**, which stores the message that the user is writing. All structures mentioned above are owned by the I-mail device driver, while the **file structure** of a user is owned by the kernel and stored in the memory, which is only accessible by the process that gets access to this user struct. In other words, the file structure is designed to be per process, not per user struct. The file structure also includes the file operations of each user node (user struct). Thus, when there is need to delete the user node, the file structure (essentially, a session) still exists and needs to free the pointer `private_data` to avoid the **dangling pointer**.

When a human user comes, he has to get access to his I-mail mailbox by entering his Linux username and password. After authentication, he can change password, check his mailbox, write a mail to someone else in the I-mail system, or simply delete the account. Because nowadays there may be several terminals running on a single Linux server, we have to guarantee that the I-mail system works concurrently or in parallel. This leads us to the lock system for I-mail. Here we take the simplest case for example, i.e., we have only one CPU accessible for the I-mail system.

We firstly consider concurrent access of different users in the user list. When user A and user B come at the same time and want to view their profile, the I-mail driver should permit it. However, whenever user A wants to send a mail to user B, he must firstly check whether user B is in the user list. Thus, a process

may be able to read data in the user list. Thus, if user B is now writing to the profile, the system crashes because the writing operation is unforeseeable to other users - if it's a deletion, then a race condition occurs and may crash the system by dereferencing to a dangling pointer. On another consideration, the reading action is much more frequent than writing, because changing the user information is rare. Thus, we consider applying a read-write semaphore on the user list.

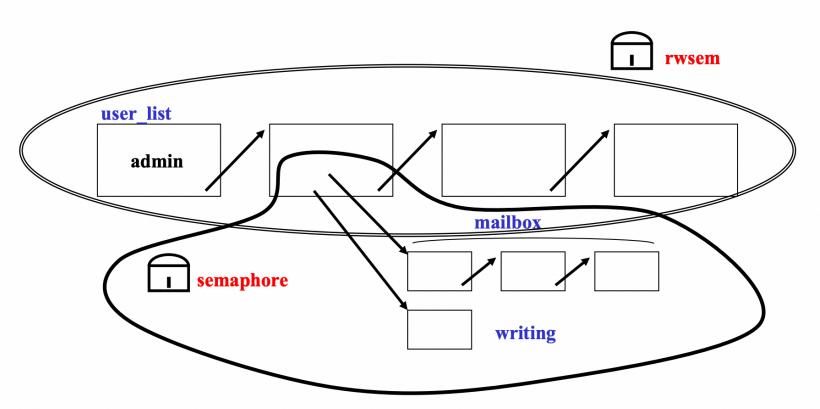

Figure 1.42: The locking strategies of the I-mail system.

Another semaphore should be applied to the structs in the user node. We can also implement a read-write semaphore here, but it's not a must, because the frequency of reading and writing to the structs in the user node should be approximately the same. Thus, the introduction to a read-write semaphore may lead to unnecessary overhead that may even overweight the normal semaphore.

We can also view the synchronization under data structures. The r/w semaphore is a semaphore that controls the `auth` field of the user structure, and controls all the user nodes in the user list. The user data semaphore is per user node and thus can be implemented as a field in the user struct, as shown in Figure 1.43.

1.10 File System

After understanding the Linux Paging system, we need to get hands on the Linux file system for further understanding how we can implement the `malloc` operation.

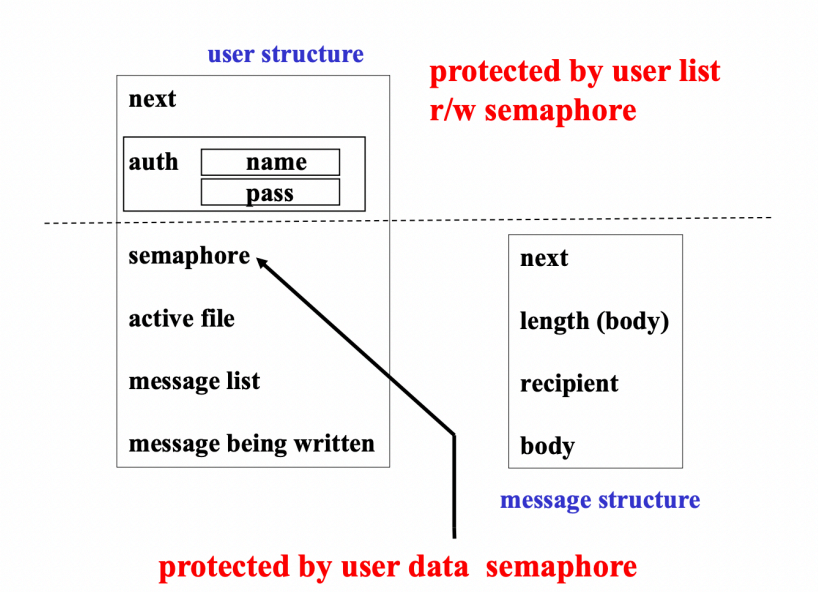

Figure 1.43: The locking strategies of the I-mail system, in a data-structure view.

1.10.1 A Block Structure

Most modern file systems use a block structure to store data and metadata[26]. Figure 1.44 shows a flat block structure which has only 1 directory, and the number of files cannot exceed 62. Each block is of 4kB size and is divided into several entries.

The first block is the **boot block**, which stores the metadata describing the whole file system. According to Figure 1.44, we have the number of directory entries (a 64B entry in the boot block), number of inodes (following the boot block), number of data blocks (the blocks storing actual data), a 52B reserved place for padding, and 64B directory entries, up to an amount of 63. Each directory entry stores represents a file, with information about the file name (`char*`), file type (normal file, executable file, directory file, ...), the inode number, and 24B reserved for padding.

The second to N blocks are used for the inode blocks. Inode is the metadata controlling which data blocks belong to this file.

[26]Metadata is data to index (find) data.

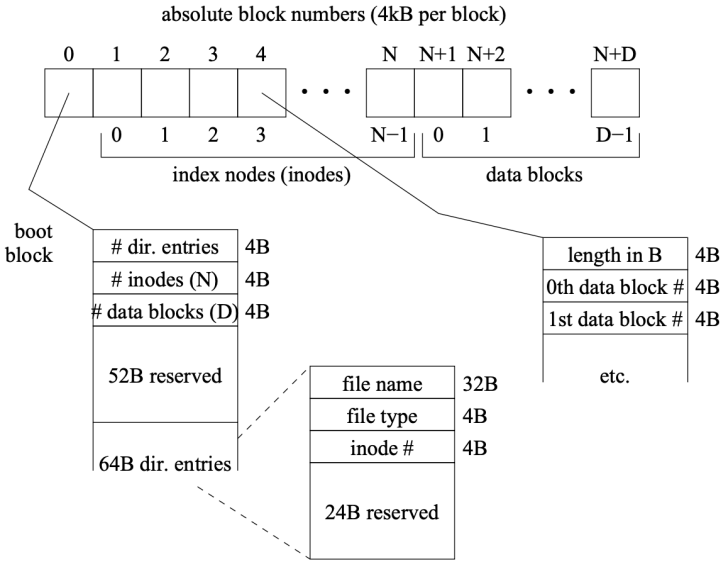

Figure 1.44: The block structure of a file system.

1.10.2 INODE

The first conception to be understood is the **inode** structure. In the physical device of a disk, data is stored in **sectors**, which is 512B (0.5K). However, reading such an amount of data wastes time due to the overhead of loading data physically, so the operating system reads 8 sectors at a time (4KB), constituting a **block**. In other words, blocks comprise the smallest unit for reading and writing data.

To get a look at the inode in a file, use command `stat <filename>`. As we see inode is metadata about data, so it's still data, and all datas need to be stored somewhere. Modern file systems achieve this when **formatting** a disk: it divides the physical memory space into 2 parts: one part for storing data, the other part storing metadata (inode).

INODE defines a file. In other words, when referencing a file on the operating system's perspective, it's refering to INODE, instead of the filename provided by the user. The operating system firstly get the INODE id using a mapping from file name to INODE, then uses the INODE id to find the INODE (node), which contains all information about the file. After finding all information about the file, we can get the file on the physical device.

This characteristic causes some strange phenomenons. Firstly, moving the file or changing the file name using `mv` does **not** change the INODE id, and thus only the path field in the INODE changes. However, copying file using `cp` **does** change the INODE number because another block of memory is allocated to store the same content but changing each does not affect the other. Secondly, After opening a file, the file system begins to use INODE id for identifying this file, and it ignores the filename since then. Thus, getting the filename from the INODE is not a usual approach.

1.10.3 Directories

In modern file systems (like Ext2), a directory is essentially a file. The structure of a directory is fairly easy: it is a list of directory entries, each consisting of the file name and INODE of a file, and all the entries combines up to all the files in the directory.

A confusing phenomenon about the permissions for a directory is, we can see that the user has `x` (executable) permission to the directory. This is because dereference INODE using the INODE id requires more permission than `w` (write).

In Linux, just like a file, a directory has an inode. Rather than pointing to disk blocks that contain file data, though, a directory inode points to disk blocks that contain directory structures. Compared to an inode, a directory structure contains a limited amount of information about a file.[27]

1.10.4 Hard Link

A **hard link** is a link directly between an INODE id and a filename. Using Linux command `ln`, we can link the filename directly to the INODE id without bothering the initial filename. Thus, when the initial file (actually filename) is deleted, the new filename is still accessible to find the file by INODE.

This is contributed by the fact that there is a field **link numbers** in INODE. When a hard link is created, this number will be incremented by 1. When a file (filename) is deleted, this number is decremented by 1. When this number becomes 0, the scavenger program will be launched and reclaim the blocks allocated for this file.

It's very important to memtion the two special files inside a directory file, namely, the file `.` and the file `..`: the first one has the same INODE id with

[27] https://www.howtogeek.com/465350/everything-you-ever-wanted-to-know-about-inodes-on-linux/

the directory file (because they are actually the same directory), while the latter one has the same INODE id with the parent directory file. Thus, we conclude that every directory has $2 + n$ link numbers where n is the number of subdirectories in this directory (not including recursive subdirectories, but including hidden subdirectories).

1.10.5 Soft (Symbolic) Link

As contrary to hard link, **soft link** is a linkage approach that's based on the original file name. If file A is softly linked to file B, then the content in file A is actually **path** to file B. In other words, if we delete file B without changing file A, file B is going to throw an error when we want to open it. We can also conclude that file B does not increase the link number in INODE.

We can use `ln -s` to create a soft link. Note that, as discussed, when we delete the original file, all soft links to it should be deleted, otherwise it constructs a file leak, as the soft link is now pointing to somewhere undefined just like an undealloced pointer.

1.10.6 Executables

A special file type in Linux is the ELF (Executable and Linkable Format) file, with magic string "elf" at the very beginning of the file. If you `cat` the file, it's likely you can find it at the beginning, and something like "file opened" and "file closed" at the end, but you'll not be able to recognize what's all the main contents in the body of the file, because they're binary.

Executables mostly imply the code part of a program. If you're trying to execute the elf file, the first thing to do is to authorize the executable attribute of this file using `chmod +x <filename>`. In other words, a file can only be executed (whether it's an elf file or not) if and only if it's authorized to be executed. Non-elf files can also be executed but as the head of the file does not contain the magic string "elf", the program will be stopped immediately, as shown in Figure 1.45.

Knowing that the elf file can be executed is only the first step for executing it. To actually execute a file, the program needs to be loaded into the operating system.[28] **Loading** is the process to move the code and data segments of the executable from the file system into the memory. For example, say I've got an ELF file `calc` and it needs a data file `score.csv`. After I type in the execution

[28]Note that linking is done in the compiling process to generate the elf file, while loading is completed when we want to execute the elf file.

```
❯ ls
README.txt graphs     text.bib   text.pdf   text.tex
❯ chmod +x text.pdf
❯ ./text.pdf
./text.pdf: line 1: fg: no job control
./text.pdf: line 2: fg: no job control
./text.pdf: line 3: 3: command not found
./text.pdf: line 80274: syntax error near unexpected token `newline'
./text.pdf: line 80274: `<< /Filter /FlateDecode /Length 69 >>'
```

Figure 1.45: Forcely execute a file that is not an executable.

command `./calc`, the operating system loads the code segment of the program `calc` into the memory and run it, and when it parsed that we need a requirement `score.csv`, so it loads the data segment of the program into the memory as well, and forms a complete program.

The program now runs with all the code and data segments in program may use up to 4GB virtual memory (i.e., the virtual memory space for each program is 4GB), the memory is likely to be used up if the data segment is large. In this case, some of the data should be **swapped** out of the physical memory to the file system so that other data in need can be swapped in (to the memory). In this case, the cache TLB needs to be **flushed** so that the content in it won't be misleading.

Chapter 2

Communicative Networks

The communicative network is also a part of kernel, so users don't bother understanding all the concepts and protocols predefined by scientists. We decide to make this chapter independent from the kernel because although it's implemented in the kernel, it has few to do with the hardware and more of communicative protocols. For those who are not interested in web engineering and communicative network, you can simply skip this chapter.

If you decide to learn from this chapter after reading you should be able to construct a TCP/IP protocol stack implementation on your own kernel.

2.1 Protocol Layers

The network protocol stack, as indicated by its name, is a pile of stacks, one on top of another. When implementing the protocol stacks, the struct is hierarchical - one wrapping up another. When sending a package, the driver encrypts the text message into an encrypted message, and with the process going down-layer, the message is encrypted layer by layer. At the bottom of the protocols, the physical devices get the final message and deliver it to another computer. When that computer receives this message, it decrypts this message, layer by layer upward, using a reversed order. The encryption process is shown in Figure 2.1.

On the top is the application layer, which is mostly constructed by user-level applications. When the user wants to send a message to somewhere (wherever the destination is), the user-space program launches a system call using the syscall interface and delivers the message to the TCP stack. The TCP stack encrypts the message and adds a TCP header in front of the application data and delivers it to

Figure 2.1: The protocol stack from physical devices to highly abstracted applications that utilizes the lower protocol stacks.

the IP stack. Similarly, the IP function calls the Ethernet devices, which appends the Ethernet data. Now the data is fully encapsulated and ready to be sent to the Internet.

2.1.1 Importance of Hierarchical Architecture

It's a common question why we need multiple layers for the network, and the answer is that each layer is in charge of one particular functionality. The **Ethernet protocol**, which is on the lowest level, is in charge of translating electric signals into data **packets**. In other words, it deals with interpretation from the physical world to the virtual world (computers). After implementation of the Ethernet protocol, the computer should be able to realize the **end-to-end communication** between two devices in the same **Local Area Network** (LAN).

However, the Ethernet is not able to communicate among different LANs. Thus, a packet can only be sent to a device that is on the same LAN as the sender, which will not be able to construct a World Wide Web as we have now. This leads to the invention of **IP** (Internet Protocol), which aims to give each LAN island an IP address so that two LAN islands can connect to each other

using this address. The process of finding another LAN island is called **routing**. The so-called **router**, which is frequently used nowadays, is based on this protocol and address. It can be understood better when you consider the router in your home: each router is designed to be with multiple interfaces, each connecting to one device. Inside each router, there is an **LRT** (Local Route Table) table defining which interface is assigned to which Ethernet address.

The IP protocol is still not reliable, because the network is prone to lose data, especially for wireless networks. Considering this, another layer of protocol needs to be implemented to ensure the consistency of the Internet messages. This protocol is called **TCP**, or Transmission Control Protocol, which is a connection-oriented, word-stream-based, and **reliable** network protocol. TCP is said to be reliable because it guarantees the data packet to be received by the recipient, which means that if the packet is lost, the sender needs to send it again and again until the timeout or the acknowledgment of receipt is sent back.

In the following sections, we discuss each protocol stack layer in detail.

2.1.2 Size and Sequence of Packets

As shown in Figure 2.1, the size of the packet in each layer is different. The **payload** of a communicative packet, or the maximum amount of the physical message packet to be sent, is fixed to be 1500 bytes. This physical message (payload) contains three pieces of information: the application message, the TCP header, and the IP header. In other words, the Ethernet header is not counted as a part of the payload, because it's a part of the translation from the physical world to the virtual world. But the TCP/IP protocols are counted as parts of the payload because they're dealing with the virtual messages.

The length of the ethernet header is fixed to be 22 bytes. In other words, the total length of a packet traveling in the physical air is a maximum length of 1522. The lengths of TCP and IP headers are both of length of 20 bytes. The overview of the packet sizes is shown in Figure 2.2.

TCP packets are sent in sequence because data is organized in sequence in a computer. Say there is a file which is of the size of 10MB. In order to send this file, TCP needs to send more than 7,100 packets in order to deliver the full message. This requires the OS to mark each packet with two **sequence numbers** (SN): the SN of itself and the SN of the next packet, from 0 to around 10*1024*1024, and then send each packet out to the Internet. Note that SN represents the number of **bytes** already sent, not simply the number of the sending packet.

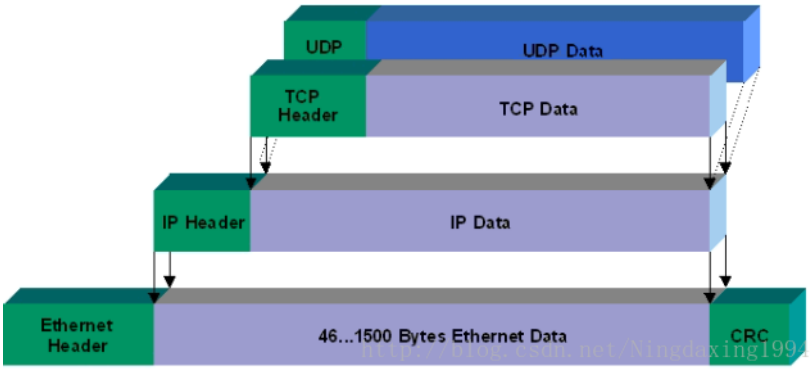

Figure 2.2: The network packet size of each layer.

2.1.3 Network Endians

Before we move on to the actual implementation of the network stack, we must note that the CPU endian is different from the network endian.

There are two types of endians: the big endine and the little endine. The big endine defines that the most significant bit of a bunch of data iss stored in the lowest address inside the memory, and the little endine is just the reverse. Figure 2.3 is a very clear explanation of the priciple.

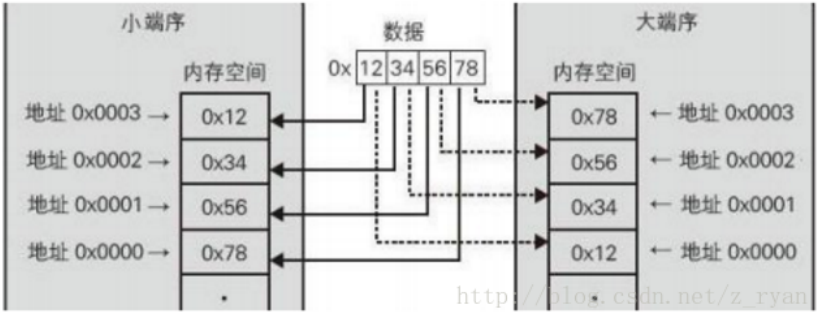

Figure 2.3: The large endine and little endine of data stored.

As we see, the data is stored in memory, consisting of a bunch of bytes. The example in the figure above takes `0x 12 34 56 78` as an example. The most significant bit of this number is `12`, so the big endine implies that "big number in the end", so the most significant bit `12` goes to the bottom of the memory, with

smallest address number (`0x0000`). On the contrary, the little endine puts the least significant bit `78` at the very bottom of the memory space, as shown in the left part of the figure.

However, this is endian in the computer memory. In networking, all the data is transferred not using a data block, but a data stream. Compared to data blocks, the data stream has no geometric features. In other words, we must reconstruct the data when we transfer the data stored in memory to the network.

However, how can we specifically define the endine in a data stream? In other words, when receiving a data stream, do we take the first byte received to be the most significant byte or least significant byte?

TCP, UDP, and IP protocols define that, the first byte received will be automatically taken as the **most significant** byte. This requires the communicative sender to send the most significant byte from the memory to the network as well. As the computer always construct data streams from smaller address in memory to larger address in memory, we conclude that the smaller address in memory should be the head of data stream, thus being received by the communicative receiver firstly, and because the first to receive should be most significant byte, we conclude that the smaller address should be of most significant byte, thus the memory is equivalent to be **big endian** in memory.

Now we focus on the implementation. First of all, it would be too troublesome if the programmer has to figure out what endine a computer has, because different computers have different endines. A practical solution would be implementing a function that transforms the endines automatically, ignoring what endine the computer have, it always transform the network endine to be correct for TCP/UDP/IP standard.

This function is called `hton` (host to network) conversion function and `ntoh` (network to host) conversion function. When the host endine is the same as network endine, the function does nothing; otherwise, it flaps every byte.

2.2 Address Resolution Protocol

The Address Resolution Protocol, or simply **ARP**, is another network protocol that lies under all the protocols. This protocol is used to solve the MAC address **given the IP address** of a device. This protocol is one of the most important protocols in communicative networks IPv4 but has been replaced by the NDP protocol in IPv6.

The Media Access Control address, or simply **MAC** address, is a unique ID of

a physical device, network interface controller (NIC), or simply called the netcard. MAC address is also called the **Ethernet address**, the **LAN address**, or the **Physical address**. As indicated by its names, the MAC port is the last port to go through when a packet goes out from the computer, and also the first port to go through when a packet goes into the receiver computer. Note that the MAC address is **unique** for each net card, which means that if there are multiple network cards on a device, the device will have multiple MAC addresses, instead of only one. The MAC address is assigned to a device when it is fabricated, so assigned when the computer is connected to the Internet, or when the network card is installed onto the computer.

Assume we now have two computer hosts in the same **network segment**, and host A wants to send a message to host B. Using the ARP protocol, we go through the message delivery process step by step:

- Host A checks its ARP table (mapping between IP address and MAC address) and determines whether it has an item of the MAC address of host B. If yes, host A will encapsulate the IP data pack into a packet **frame** and send it out to host B, using the MAC address found in the ARP table. This is the simplest and most efficient process of sending a packet.

- If host A cannot find host B in the ARP table, it will cache this datagram (from the IP protocol layer) somewhere, and **broadcast** an ARP request to all members in this network segment. The ARP request contains the IP address and MAC address of host A, and the IP address of host B. Note that because the MAC address of host B is not yet available **to host A**, the ARP request from host A does not contain the MAC address of host B. This field is filled with all 0's instead.

- Because the ARP request is broadcasted, all hosts in the network segment will receive this request. However, only host B confirms that the MAC address in the ARP request is the same as its MAC address, so only host B is going to make a reply to host A. Host B saves the IP address and MAC address of host A, and then **reply** to host A end-to-end with a response, with the MAC address of host B filled with the correct value.

- After host A receives the response from host B, it fills its ARP table with the IP address and MAC address of host B, and subsequently sends the cached datagram to host B after encapsulation.

2.3 Internet Protocol

The Internet Protocol (**IP**) is the protocol above ARP. To understand this protocol, the most important structure to grasp is the header of the IP datagram. Figure 2.4 is a very clear implementation of the IP datagram, from which we're going to learn from.

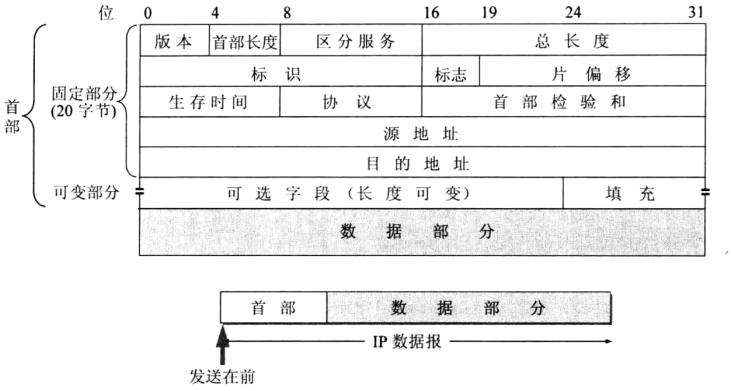

图 4-13 IP 数据报的格式

Figure 2.4: The format of the header of the Internet Protocol.

The first 4 bits are called **version number** bits, which indicate whether the IP protocol is version 4 (IPv4) or version 6 (IPv6). When two hosts want to connect to each other, they must have the same version number bits.

The second 4 bits are called **header length** bits, with a unit of 4 bytes (a **word**). Because this field only has 4 bits, the header length must be a number between 0 byte and 60 bytes. Because the minimum length of the header is 20 bytes, the minimum header length bits will be `0101` (5 words). As the unit is word, when the header length is not a multiple of a word, there will be padding bits to fix this issue.

The next 8 bits are **DiffServ** (Differentiated Services Code Point) bits, which we will not cover according to the scope of this book. However, note that this field takes one whole byte.

The next 16 bits are called **total length** bits, which represents the total length of the datagram, including both the header and the actual data part. Note that the unit of the total length bits is not a word, but a single byte. Thus, the maximum

length of the total length is 65535 (also called the **MTU**, Maxiumum Transmission Unit). When this does not satisfy to represent the whole data stream, the data is cut into slices and be delivered in sequence.

On the second line, the first 16 bits are called **identification** bits. The identification bits actually constructs a counter. Every time when a datagram is generated, this field is incremented by 1. However, note that IP is a **connectionless service**, which means that when the host delivers a datagram, it has not established a connect with the receiver. Thus, when one data stream is cut into slices, it's not possible for the receiver to give a sequence to the counter. Instead, when we have sliced data, all the slices of this data stream have the same identification bits. Thus, when the receiver gets the slices, it can reconstruct the data stream by finding the same value in the identification field.

The next three bits are called **flag** bits, which is constructed by the More Fragments (MF) bit (least significant bit), the Don't Fragment (DF) bit (middle bit), and the Reserved bit (most significant bit). The MF bit is used to represent whether there are more datagrams following this datagram. If MF is 1, the answer is yes. The DF bit is used to represent whether we can slice (fragment) the data stream, note that when DF is **0** the data stream can be fragmented, **not** 1.

The last 13 bits in the second line are called the **Fragment Offset** bits, which are used to indicate the relative position of this fragment in the original datagram. For example, assume the original datagram is 3*65535 bytes. Thus, we need 3 full fragments to transmit this datagram. However, note that this number is **not** the number of fragment, but again, the position of this fragment. In the example above, the fragment offset for the first fragment should be 0, for the second fragment it should be 65535+1, and for the last fragment it should be 2*65535+1. However, note that the basic unit for the fragment offset field is 8 bytes (a **quad**). Thus, the actual values should be 0, 65536/8 (8192), and 65536*2/8 (16384). Again, if the last fragment has a length that is not a multiple of 8 bytes, a padding should be applied to it.

The first 8 bits in the third line are called the **Time To Live** (TTL) bits. This is a crucial concept in the modern communicative network, and can be frequently seen. The TTL represents the maximum number of leaps after it's transmitted. A **leap** means a walkthrough over a router (a LAN). Every time when the datagram travels through a router, this field is decremented by 1. When this field becomes 0, the last router will **discard** this datagram. Because we only have 8 bits for this field, the maximum TTL is 255.

The second 8 bits in the third line are called the **Protocol** bits, which represent what protocol the datagram follows. There are more than 200 protocols, but what

we use most are the TCP protocol and UDP protocol.

The last 16 bits in the third line are called the **Header Checksum** bits, which are used to check whether the datagram is correctly delivered. The implementation of this algorithm is shown in Figure 2.5 in detail.

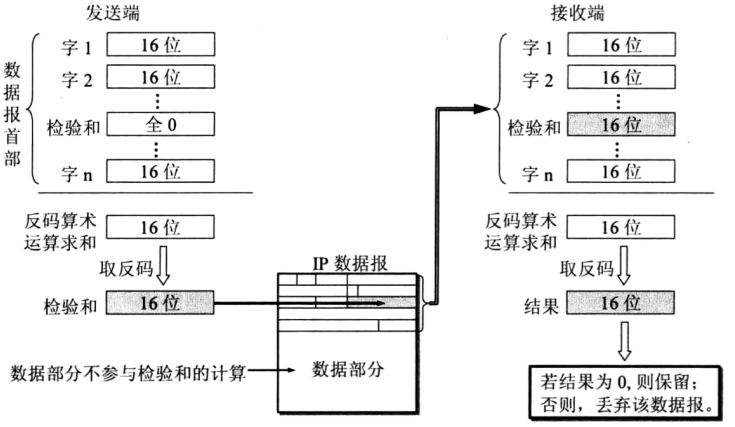

图 4-15 IP 数据报首部检验和的计算过程

Figure 2.5: The principle of the checksum function.

Firstly, note that only the datagram header participates in the checksum calculation. In other words, the checksum value will not change if the data part changes. Secondly, when sending the datagram, the whole checksum field is placed to be 0. Then we calculate the sum of all the one's complements of all the bytes in the datagram header, and place the result into the checksum field. The datagram is now ready to be sent to the receiver.

Then the receiver gets the datagram. The receiver calculates the sum of the one's complements of all the bytes in the datagram, and decide whether the result is 0. If it's equal to 0, the datagram is valid. If not, the datagram must be changed during transmission, thus should be discarded.

The next two lines are both used for IP addresses. The fourth line is used for the source address, and the fifth line is used for the destination address. These two addresses are used to identify the source subnet and destination subnet IP addresses.

2.3.1 Subnet and Subnet Masks

IP addresses are precious, according to the IP format `a.b.c.d`, where each field is a number with less than 3 decimal digits, we can have at most 10^{12} IP addresses in the world. However, the number of machines with network is much larger than this number, while lots of IP addresses can be combined together, because they do not connect to the Internet. Thus, **subnets** are invented to split blocks of networks so that they share the same IP address. The subnet is also called a **LAN** (local area network), which is typically constructed by a **router**, as shown in Figure 2.6.

Figure 2.6: The three types of network interfaces on a router.

The LAN interface is used to connect to all the user machines in the local area, the **WAN** interface is used to connect to the **modex**, which transforms the light signals to electric signals. When we consult the machine IP, there are two possible results: **public IP** and **private IP**. The public IP is the IP address that can be accessed by all the machines on the Internet, and the private IP is the IP address that's only valid inside the subnet (LAN), and can be consulted using the command `ipconfig` on a desktop computer. To distinguish between the public IP and private IP, the IP committee also invented a protocol for the IP address format, with class A to class E. Here we introduce the most commonly used classes: class A to class C. Each class differs in the format of the IP address and the subnet mask they have. The segment address ranges of all the classes are shown in Table 2.1. From the table we conclude that we can always use the first 8 bits (0-255) to classify which class the IP address belongs to.

IP Type	IP Start	Subnet Mask	Segment Start	Segment End
A	N.n.n.n	255.0.0.0	1.0.0.0	127.255.255.255
B	N.N.n.n	255.255.0.0	128.0.0.0	191.255.255.255
C	N.N.N.n	255.255.255.0	192.0.0.0	223.255.255.255

Table 2.1: The three most commonly used IP address types and their subnet masks. **N**: network bits; **n**: node bits.

The **network mask** is a series of bits to indicate what bits in the IP address are not considered to be shown on the Internet, where the number of bit 1's will be the number of **network bits**, and the number of bit 0's will be the number of **node bits**. More precisely, because the subnet mask has exactly the same format as the IP address, each bit 1 of the subnet mask represents that the IP address bit at the same place is a network bit. For example, if we have the subnet mask to be 255.255.0.0, then the first 16 bits are for network bits, and the last 16 bits are for node bits. Similarly, 255.192.0.0 means that the first 10 bits are for network bits. The number of network bits has a fancy name: **CIDR** (Classless Inter-Domain Routing), which is often attached to an IP address. For example, the IP/CIDR address 192.168.4.1/28 means that the IP address of the machine is 192.168.4.1, while the last 4 bits (0001) are node bits, while the first 28 bits are for network bits. The address format, subnet mask, and the start and end of the **private network** addresses are shown in Table 2.3. If the IP address is in the range of "Start" and "End", then the IP address must be a private IP address, otherwise it must be a public IP address.

IP Type	Address Format	Subnet Mask	Private Start	Private End
A	N.n.n.n	255.0.0.0	10.0.0.0	10.255.255.255
B	N.N.n.n	255.255.0.0	172.16.0.0	172.31.255.255
C	N.N.N.n	255.255.255.0	192.168.0.0	192.168.255.255

Table 2.2: The three most commonly used IP address types and their subnet masks. **N**: network bits; **n**: node bits.

The network bits are used to identify which subnet the machine is in, while the node bits are used to clarify which machine it is in the subnet. In a subnet, all the IP addresses of one machine must be different from another machine, while they

share the same public IP assigned by the network provider (**IANA**, Internet Assigned Numbers Authority). Thus, there is **no** one-to-one mapping between a private IP address and a public IP address - all private addresses map to one public IP address. When a subnet machine sends a packet to the outside world, the packet is firstly delivered to the router commanding this subnet, then forwarded to the Internet after a private-public transformation.

2.3.2 Gateway, Router, and NAT

As it's not sufficient to know only about the IP address strategies from a high level, we introduce the hardwares to support these strategies. This section provides a clear view of the devices used in our daily life.

An important role in the private-public transformation is called the gateway, which is often realized by a router. The **gateway** is the abstract device that contains 2 network cards: one with MAC address corresponding to the public IP address, the other with MAC address corresponding to the private IP address. These two network cards provide a service called **proxy**, which translates between the two addresses. The reason we call the translator a proxy is that, the process of translation is transparent to the packet sender. For example, when the sender wants to send a packet to a machine in the subnet, it does **not** need to know what the translation is - all that is needed is to know the public IP address of the gateway (proxy of the subnet). The proxy guarantees that, after the sender delivers a packet to the proxy, the packet will always translate the address and forward to the correct machine the sender specifies. This service is also called **NAT** (Network Address Translation), which is illustrated in Figure 2.7.

There are three main types of NAT: static NAT, pooled NAT, and Port-level NAT (NAPT). The most commonly used one, NAPT service, can map a private IP address to a **port** in the shared public IP address. If we take Figure 2.7 for example, where we clearly see that we have one public IP address for 3 private IP addresses, we can guarantee that the mapping will fail because all packets will be broadcasted to all machines in the subnet. Using NAPT, we can track not only the private IP address but also the port of the private IP address. We take an example as shown in Figure **??**. When two machines send an IP packet to the server at the same time, the router firstly looks at their IP port. Because they have the same port 4096, the router automatically assign the later packet with a port number incremented by one (4097), because each port can only be used once when we have only one public IP address. Thus, when the server sends back the packet, the information already indicates the port, thus the router is able to back-translate the private IP address from the public IP address with a port

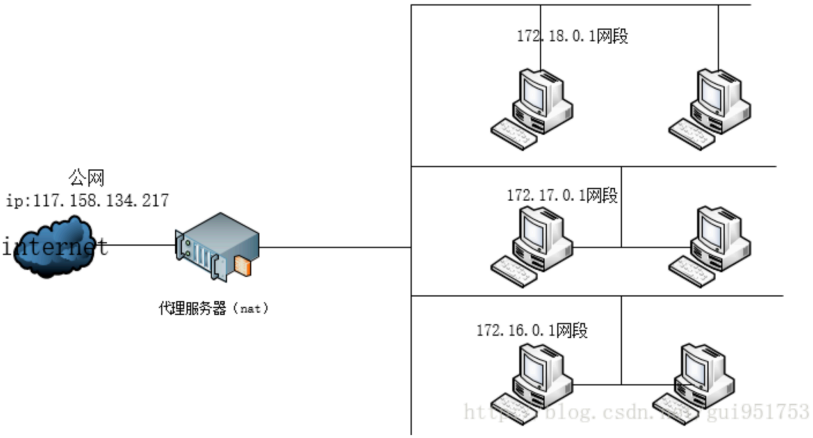

Figure 2.7: The IP address mapping using NAT.

specified.

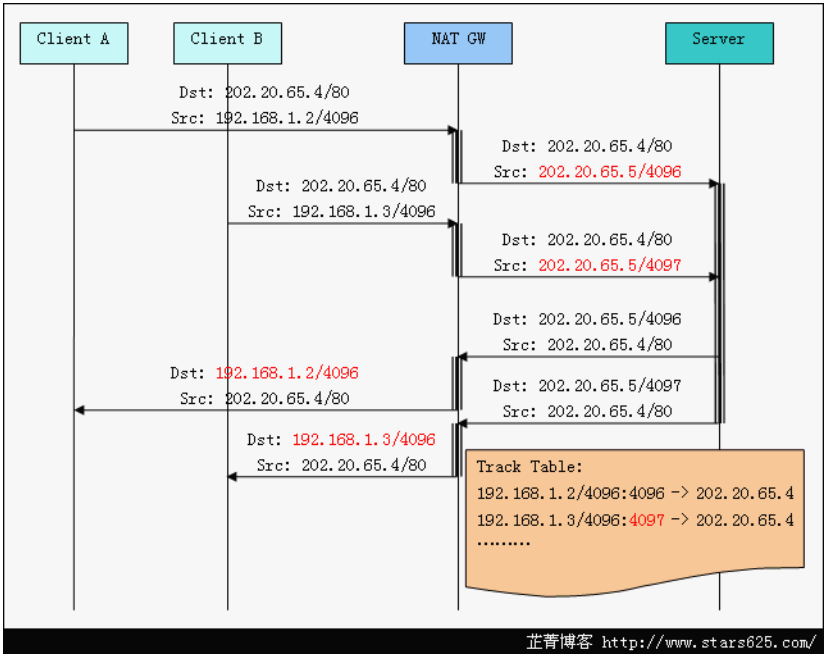

Figure 2.8: The IP address mapping using NAPT.

2.3.3 Ethernet Switch and Router

The Ethernet switch and router are two frequently mistakenly mixed concepts. These two devices are essentially totally different, because the Ethernet switch works on Ethernet layer, while the router works on the IP layer. However, they do have the same purpose: forwarding network packets.

The router, as mentioned in the last section, is used to forward **IP** addresses. When the subnet is connected to the Internet, it needs a router as a proxy to be integrated into the Internet, so the machine is able to be "online". Thus, a router works in the Internet and plays the role of a gateway for network packets to enter or exit a subnet from or to the Internet.

The Ethernet switch, however, does not provide any information about the IP address. The only functionality of the Ethernet switch is to forward data inside the subnet by changing the **MAC** addresses of the source or destination of the packet in transmission. In other words, the Ethernet switch is used in a subnet, so it works under the router.

However, note that the router often integrates several LAN ports, which has essentially done the functionalities of the Ethernet switch. On the other side, the Ethernet switch does not contain any functionality of the router, so it cannot work independently without a router providing the subnet.

We'll go through an example to distinguish these two devices better. Before the network packet is sent out from your machine, the source MAC is the MAC address of the Ethernet switch, and the destination MAC is the MAC of the Ethernet switch. After the network packet goes through the switch, the source MAC address in the Ethernet frame is updated to the MAC of the switch, and the destination becomes the router. This behavior is because each gateway in the network provides a transparent to the upstream, which forms a proxy, as mentioned in the last section.

	Ethernet Frame		IP Datagram		TCP Segment	
	Src MAC	Dst MAC	Src IP	Dst IP	Src Port	Dst Port
Machine	Machine	Switch	Machine	Server	6879	80
Switch	Switch	Router	Machine	Server	6879	80
Router	Router	Nextnode	Subnet	Server	2503	80

Table 2.3: The Ethernet frame, IP datagram and TCP segment of the network packet after the network packet going out of the machine, going through the Ethernet switch, and going through the router.

After the router translates the network packet, the source MAC address becomes the router, and the destination MAC becomes the next node of router in the Internet, decided by the router itself. The important note here is that the source IP address specified in the IP datagram also changes to the *public IP* of the subnet defined by the router. Before going through the router, the source IP was the private IP of the machine *in the subnet*. Another note is that the source port of the TCP segment also changes due to the NAT service, as discussed in the last section.

2.4 Transmission Control Protocol

The **Transmission Control Protocol**, or simply **TCP**, is the most important and most frequently examined protocol among all the OSI network protocols. It is very important in modern network systems, because it contains the so-called "3 handshakes" to establish a reliable connection between the communicative sender and receiver. In this part, we'll still focus on the segment header of the protocol, and try to understand how each field works. The segment header is shown in Figure 2.9.

TCP segment header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																	Urgent pointer (if URG set)														
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
60	480																																

Figure 2.9: The segment format of the TCP header.

The first line is split into two 16-bit fields, one representing the **source port** and the other representing the **destination port**. This format is pretty similar to the IP header, so we don't bother repeating it here once again.

The second line is monopolized by the 32-bit **Sequence Number** (SN) field, which represents the order of the fragment in the datagram, as already explained

in Section 2.1.2.

The third line is monopolized by the 32-bit **Acknowledge Number** (AN) field, which we'll discuss later.

The fourth line is for a series of settings, as shown in Figure 2.9. The **URG** bit is used for marking whether the urgent pointer (on the fifth line) is valid. The **ACK** bit indicates whether the Acknowledgement number (on the third line) is valid. The **PSH** bit should force the receiver read the datagram from the TCP cache.

The **RST** bit asks for resetting the connection. We call the segment with RST bit turned on the **reset segment**. The **SYN** bit is used for requesting to establish a connection, so the segment is called **synchronization segment**. The **FIN** bit is used to notify the receiver that, this host (transmitter) is going to shut down, so the segment is called **ending segment**.

The first 16 bits in the fifth line is used for checksum, which is the same as IP header. However, the calculation of this checksum includes both the TCP header and the TCP data body.

2.4.1 The Three Handshakes

To establish a reliable connection between the sender and receiver, the two ends must make sure that the other is open to its network packets. Because IP does not provide such (reliable) connection, TCP has to deal with this issue. The way that TCP establishes the connection is by the so-called **three handshakes**, as shown in Figure 2.10.

Before the establishment of connection, the two ends (client and server) are both closed to each other. Before it's possible to establish the connection, the server must construct a **TCB** (Transmission Control Block), and turn itself into a listening status.

Now the client wants to send a message to the server. The client firstly construct a client TCB, then send a **connection request segment** (synchronization segment) to the server, with the SYN bit to be 1. The segment also contains the SN to be randomly chosen, say value x. Now the client process goes into the SYN-SENT status, which means that the synchronization segment has been sent. Note that this segment does **not** contain any actual data, but still takes one SN. This process is called the **first handshake**.

There is possibility that the server does not get the sync segment. If this is the case, the client will send the same request to the server repeatedly, until timeout.

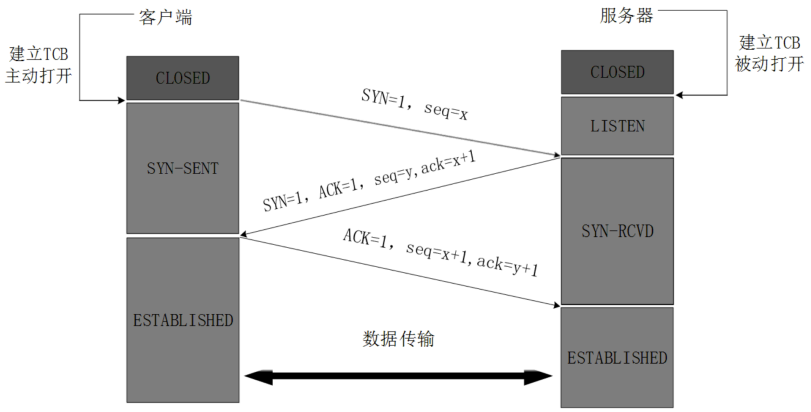

Figure 2.10: The three handshakes of the TCP protocol.

However, if the server gets the sync segment, it will decide whether it agrees with the request. If it agrees, it sends back a **confirmation segment**. The ACK and SYN bits are both 1 in the confirmation segment, and the AN is $x + 1$. Because each segment should have an SN, we randomly pick SN to be y. Now the status of the server turns from LISTEN to SYN-RCVD. Similarly, this segment cannot contain any data, but also takes one SN. This is called the **second handshake**. Up to now, the synchronization process is already finished.

If the client receives the confirmation segment from the server, it needs to send another confirmation segment back to the server. Because the sync process is already completed, the SYN bit should be 0. As this is the acknowledgement of the last confirmation segment, the ACK bit should still be 1. SN needs to be $x + 1$, and AN should be $y + 1$. After **sending** the second confirmation segment, the client is already in the ESTABLISHED status. After **receiving** the segment, the server will enter the ESTABLISHED status. Now the connection is all set. This is called the **third handshake**.

2.4.2 The Four Handwaves

After all data has been delivered, TCP needs to shut down the connection to ensure security. Before even trying to make a handwave, both the client and server should be in status ESTABLISHED. The graphical illustration is shown in Figure 1.1.

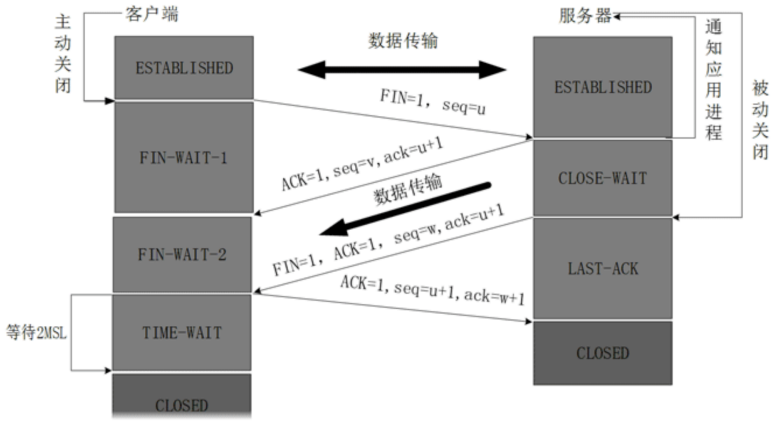

Figure 2.11: The structure of the TCP pseudo header.

The first handwave is that the **client** sends a **finish segment** to the server. After this segment is sent, there should not be any more segments with data being sent. This finish segment should contain FIN with value 1, and SN value u, where $u = b + 1$, and b is the number of bytes of the data sent. After the finish segment is sent, the client goes into the FIN-WAIT-1 status. Again, although this segment does not contain actual data, it still takes one SN.

In the second handwave, after the server gets the finish segment, it sends back a **confirmation segment**, with ACK value 1, AN value $u + 1$, and SN value randomly picked to be v. Now the server enters the CLOSE-WAIT status. In the meantime of sending the segment, the server notifies the higher level applications that the client has entered the **half-closed** status, during which the client will not be sending any segments, but the server can still send segments, and the server knows that the client will accept it.

After the client receives the confirmation segment, it enters the FIN-WAIT-2 status, and waits for the server to send the connection release segment, which indicates that all the data has been sent. Before this segment is sent, the server is still able to send segments with actual data, but the client cannot send any more segments with actual data.

The third handwave is essentially the delivery of the **connection release segment**, which is sent from the server to the client after all data segments have been sent. The connection release segment has FIN value 1, AN value $v + 1$, and SN value w. Note that if there is no actual data segments sent during the CLOSE-

WAIT status of the server, then we have $w = v$. After this segment is sent, the server goes into the LAST-ACK status and can no longer send data segments.

The fourth handwave happens when the client receives the connection release segment. After the connection release segment is received, the client should reply another confirmation segment immediately, but it shuts down a bit later. It firstly goes into the status TIME-WAIT, and after a time lag of $2s$ it is totally shut down and enters the CLOSED status. Note that s is the Maximum Segment Life (MSL) defined in TCP. This confirmation segment contains ACK value 1, AN value $w + 1$, and SN value $u + 1$. After the server gets this confirmation segment, it shuts down immediately, so it's common that the server shuts down a bit earlier than the client.

A common question is, why there are three handshakes but four handwaves. The answer is that, after server receives the handwave packet from the client, it may still have data to send to the client, and that causes a time lag.

2.4.3 TCP Checksum

The checksum rules of TCP is different from IP. In IP, only the header is taken into calculation, but in TCP, the whole segment is taken into consideration. However, the principle is the same as IP: the sender encodes the checksum and the receiver decodes the checksum. TCP checksum sums up all words (2 bytes), and the range includes both header and body.

To make use of the TCP checksum, a struct called pseudo-header should be implemented. The **pseudo-header** of a TCP segment is extracted from the IP datagram, aiming to enable TCP to check whether the data has successfully arrived at the destination. The pseudo-header is only used for checksum. The structure of the pseudo-header is shown in Figure 2.12.

Much like the IP header, the TCP pseudo-header contains the source address, destination address, mbz and protocol type. The only difference is that the pseudo-header also contains the TCP length, which is the sum of TCP header and TCP data.

Here we discuss the algorithm of checksum. Firstly, when initialization, we clear the checksum field to 0. Then we cut the pseudo-header, header, and data parts all into words (16 bits). We add all up and add the overflowed bits to the least significant bit (i.e., recurrent sum), and at last take the one's complement of the result. Then the segment is sent to the receiver.

When the receiver gets the segment, it adds up all the fields and see whether the result is all 1's. If the answer is yes, then the checksum is correct, and the

TCP pseudo-header for checksum computation (IPv4)

Bit offset	0 – 3	4 – 7	8 – 15	16 – 31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length
96	Source port			Destination port
128	Sequence number			
160	Acknowledgement number			
192	Data offset	Reserved	Flags	Window
224	Checksum			Urgent pointer
256	Options (optional)			
256/288+	Data			

Figure 2.12: The structure of the TCP pseudo header.

segment will be accepted.

2.4.4 Sockets

As some readers may have concerned, when there are multiple processes sending or receiving data streams, TCP is likely to crash. For example, when we have a server receiving a segment with SYNC to be 1 and SN to be x_2, after which we received a segment from another client, with SYNC to be 1 and SN to be x_2, the server will be confused about the behavior and take the input as invalid segment.

To address this issue, the socket is invented. A **socket** is a structure that differentiate multiple connections with multiple processes for communicative networks, and is especially useful in IP and TCP. In other words, the socket system is the network equivalence of multi-threading system for the processors.

There are three main fields in a socket: the **destination** IP address, the transmission-level **protocol** (TCP/UDP), and the **port** being used. Note that the source IP address is not a must, because it's always the localhost (this machine). The workflow of the socket system is shown in Figure 2.13.

Firstly, the server creates a socket. When initializing, the server checks the IP address type (IPv4/IPv6), the protocol type (TCP/UDP), and the socket type then makes a decision of what socket to create.

Secondly, the server binds an IP address and port for the socket. The **bind** process means to assign permanent information for one socket.

Thirdly, the created socket listens to the port and decide whether there is a connection request sending in. Note that hitherto, the socket is **not** yet opened. In

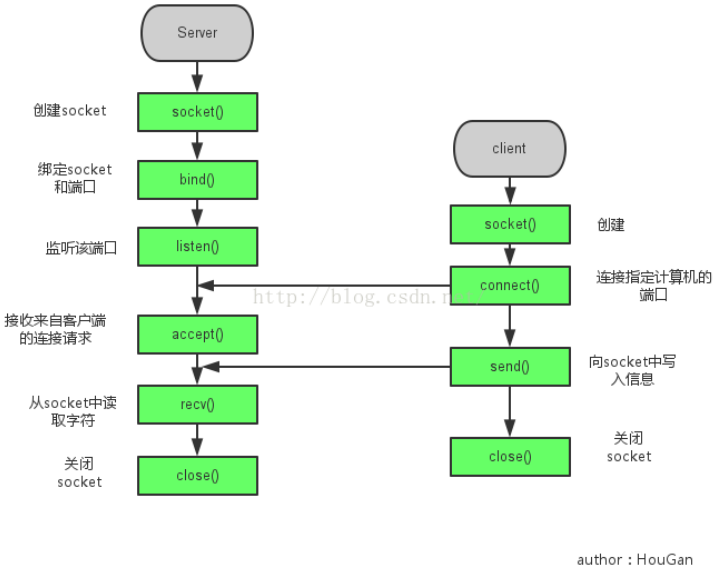

Figure 2.13: The multi-stream workflow using sockets.

other words, a listening socket may not be open. Now the server setup is finished, and it waits for the client to send a request.

For the client, it firstly creates a socket just like the server. Then it **opens** the client socket, and uses the given IP address and port to connect to the server socket.

After the server socket receives the request, it is now opened and ready to receive the requests from the client socket. Note that after the server client is opened, it goes into a **blocked** status, which continuously waits for a message to be sent from the client socket.

After the client completes connection, it sends the **connection status information** to the server socket, and then the `accept()` blocking function in the server socket returns. This indicates that the connection is now successful.

Now the server socket is able to receive the information sent from the client socket. After the socket receives the information, the client closes the connection, and then the server closes the connection.

2.4.5 Buffers: Full Duplex

When creating a TCP socket, the operating system creates two TCP buffers at the same time: one transmission buffer and one receiver buffer. When a TCP packet (larger than one TCP packet capacity) needs to be sent out, the data stream is sent out batch by batch, and each batch takes one buffer. Thus, the buffer is always full until the last batch. When the TCP packet is too short, it will wait in the buffer until nearly fill the buffer, to avoid a waste of TCP packet capacity.

When receiving TCP packets, similarly, the operating system will buffer the data read from the network card, and the applications can only get the data from the buffer provided by the operating system, not directly from the network card.

As we see, TCP has both buffers: one for transmission, and the other for receiving packets. Thus, for this connection, the host can read and write data at the same time. This system is called a **full duplex** system.

For comparison, there is a type of system called **half duplex** system. This type of system can only can do one thing from reading or writing at the same time. For example, when there is only one data line, the data stream can only goes from A to B, or from B to A. Only when we have two lines, we can have reading and writing going at the same time.

2.4.6 Applications

Almost all modern user-level applications are based on TCP, instead of UDP. For example, HTTP/HTTPS, SSH, Telnet, FTP, and SMTP. We'll go through each one and have a simple understanding of each user-level application in a later section.

MARKED...

2.5 User Datagram Protocol

Another transmission-level protocol, as a counterpart to TCP, is the User Datagram Protocol, or **UDP**. Compared to TCP, UDP is an **unreliable** transmission protocol, which means that it may lose packets, but it's also faster than the TCP protocol.

UDP has some advantages compared to TCP. Firstly, as already mentioned, UDP is much faster than TCP. In order to construct a reliable connection, TCP has to maintain the states and wait for signal segments back and forth between

the host and client, but UDP gets rid of all that, so it's both time efficient and spatial efficient. DNS uses UDP because DNS needs a great performance on time.

UDP is datagram-oriented, not segment-oriented (or data stream-oriented), which means that UDP does not change anything to the datagram from the IP layer or application layer except adding a header. The UDP datagram header is shown in Figure 2.14.

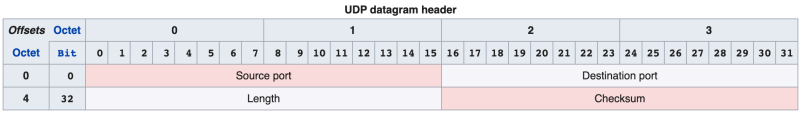

Figure 2.14: The format of the UDP datagram header.

Compared to TCP, UDP has a very regular header. The header has 8 bytes (4 words), and each word represent one field. The first field is the source port field, and is only utilized when a response is needed. The second field is the destination port, which is used when the receiver wants to deliver the datagram. The third field is the total length of the UDP datagram, including the header, so it has a minimum number 8 (bytes). The last field is the checksum field, which is also optional. We'll not look into the checksum field in UDP, as it's essentially the same as the TCP checksum, with a pseudo header and decoder-encoder architecture.

2.6 HTTP/HTTPS

The HTTP/HTTPS is the most famous application-level protocol constructed on TCP as its transmission-level protocol[1]. With the HTTP protocol, HTML pages can be generated, transmitted, received, and also parsed. It is the base for all the browsers we see nowadays.

HTTP, or **HyperText Transfer Protocol**, is a protocol defining the standard between communication of the server and client. In the context of HTTP, the server is the website, while the client is the terminal user, which can be a browser, a scrapy program, a RESTful connection end and so on.

In the terminology of HTTP, we call the server an **origin server**, and the client a **user agent**. When the user agent sends a request to the origin server, the request notifies the server that the user agent wants to establish a TCP connection

[1]Actually, HTTP can construct on other **reliable** transmission-level protocols, but we take TCP for simplicity.

with the server. When the server receives the request, it sends a **status** and a **response** back to the user agent. The status may be `200-OK` or `404-ERROR`, and typically construed of a status code and status text. The response can be anything following the HTTP protocol, like text, files, error message, and so on.

HTTP does not need to worry what the server should do when the connection fails, because this is already guaranteed by its transmission-level protocol - TCP. Thus, the OSI model provides a good task management for each level, so that each level can focus on its own task.

HTTP has some features. Firstly, it is a protocol based on the request-respond model. In other words, if there is a request, there must be a corresponding response; if there is a response, there must have been a request that triggered this response. There will only be one response to a request, and a request and only trigger one response.

Secondly, when the user client requests a service from the origin server, it only needs to give a **method** and a **path**. The most commonly used methods are `POST`, `GET`, and `HEAD`. Each methods represents a different **association** between the user client and the server origin. The path identified in the user client request is used to identify what resources to request.

Thirdly, HTTP supports transmitting **any** kind of data, and the data type is identified by the **Content-Type** field.

At last, HTTP is non-connection and non-stateful. Non-connection means that HTTP does not support a connection session, thus after one request and respond pair is transacted, the TCP connection is immediately cut off. Non-stateful means that the protocol has no memory of what it did, due to lack of states. Thus, if a later request contains a request to a resource that was already given, the server origin does not know it and has to deliver it again.

All the files stored on the Web servers are HyperText content. In other words, if the client want to get resources from the Web server, it must follow the HTTP protocol as well. If you use a web scrapy or web browser, you've already using followed the HTTP protocol, because these tools utilizes the HTTP protocol to communicate with the server origin.

The text we put into the website space in the browser is called **Uniform Resource Locator** (URL). Each URL has a unique IP address, and URL can either be a domain or an IP address. The URL also acts as the path you deliver in the request. The format of the HTTP request and response is shown in Figure 2.15.

MARKED...

图 15-4 HTTP 请求报文

Figure 2.15: The format of the HTTP request and response.

2.7 Domain Name System Protocol

The Domain Name System Protocol, or **DNS Protocol**, is the protocol to parse the domain address into an IP address. The so-called **domain** is a path on the Internet, split by periods ("."), and is of variable length. Thus, it is not good for computers to recognize, but good for users. On the other hand, the IP address is fixed in length, but hard to remember.

The mapping between domain address and IP address is stored in a file called `hosts`. Before DNS protocol was invented, Internet uses NNIC (Internet Network Information Center) to keep the `hosts` file, but soon found it expanding larger and larger, thus slower and slower to look up. Thus, the DNS protocol is invented to address this problem.

The DNS server is a system management agency in a system, which means that one system only needs one DNS server. If a new computer joins the system, it needs to add its own entry of the mapping of the domain address and IP address in to the `hosts` file. Thus, when the user agent enters a domain, the DNS service will automatically translate the domain address into the corresponding IP address defined in the `hosts` file.

As mentioned, the domain is a string consisting of several periods. The string after the last period is the top domain, and with the string going right, the domain level is downgraded gradually. For example, in the domain `ncsa.illinois.edu`,

`edu` is the top domain, which means that the domain is an educational institute; `illinois` means that the domain is for the company/organization called Illinois, and `ncsa` means that the institute of the organization is called NCSA. Note that each level of domain has a DNS server, and the number is always larger or equal to two. The reason is that the domain is for hierarchical use, and more servers include more fault tolerance. Each DNS server knows the IP address of the lower-level domains.

Here we go through the basic steps of DNS parsing. After the user client enters the domain and sends the request, the DNS server firstly looks for the DNS server on the same level. For example, if the user client has domain `ncsa.illinois.edu`, and it wants to reach out to the domain `csail.mit.edu`, the DNS system firstly dispatch the request to the DNS server at domain `illinois.edu`, but it cannot find the requested domain. Thus, the system will go to the upper level again, and look for the mapping. In this example, the system can find the mapping in the top-level DNS server in domain `edu`, and thus return the mapping. The system has the highest domain `/` (**root domain**), which has all the domains inside. Note that the root domain is different from the top domain. The root domain is even higher than the top domain because it has the mapping for all domains in the world.

During the recursively searching process, the system will cache the mappings if a request is made.

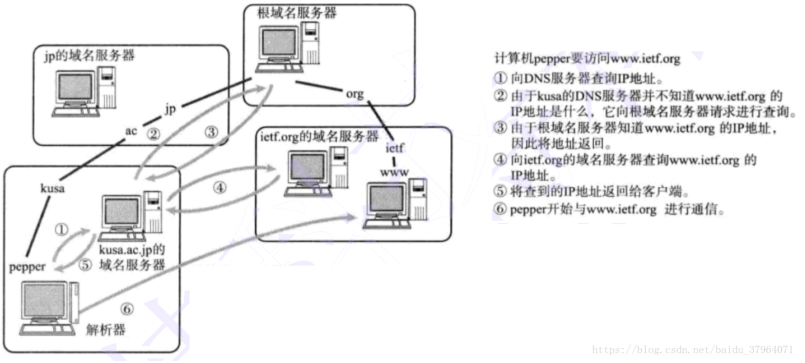

Figure 2.16: The search process of a DNS service.

Here we look into the format of the DNS request and the implementation of parsing the domain string. The format is clearly shown in Figure 2.17.

The first word in the header is called **identification**, which is a unique number for each DNS request. For example, when the user client sends a DNS request to

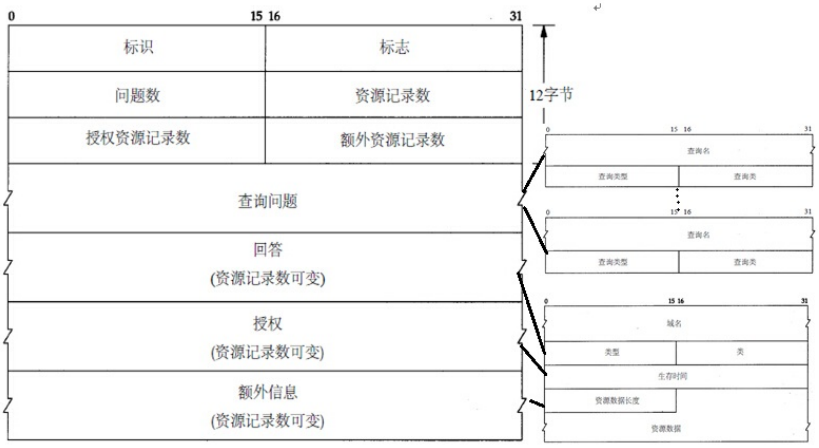

Figure 2.17: The format of the DNS header.

the DNS server, the server must return a DNS respond with exactly the same ID.

The second word in the header is called **flags**. This field is used for a lot of utility flags, as shown in Figure 2.18. QR is used to identify whether this message is a request (0) or response (1), opcode is used to set the type of query, including 0 (standard query), 1 (inverted query), 2 (server status query), and all other bits are reserved. Other flags will be illustrated later.

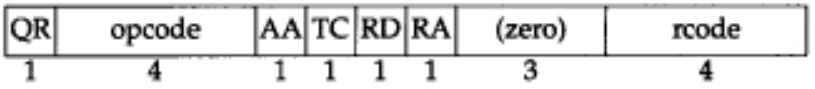

Figure 2.18: The format of the flags field in the DNS header.

The next fields are for query number, answer number, authorization number, and additional number.

Index