第4章 二级分页机制

> 文无定法。

写在前面

在本章中，同学们会了解到如下知识。

- C语言可变参数的使用和原理。
- printf的实现原理。
- 分页机制的思想。
- 二级页表分页机制的使用。

参考资料

- 可变参数函数详解
- 《操作系统真象还原》

printf的实现

在第3章，我们已经将向屏幕输出字符的函数封装在了类 `STDIO` 中。相比于使用gdb，我们更倾向于简单地使用cout或printf来debug。同时，我们之前封装的函数过于简单，这样的输出函数显然是远远不够的。因此，我们接下来就要实现一个能够进行基本的格式化输出的printf函数。但在此之前，我们需要了解C语言的可变参数机制。

可变参数

对于printf这类函数来说，我们可以使用任意数量的参数来调用printf，例如

```
printf("hello world");
printf("this is a int: %d", 123);
printf("int, char, double: %d, %c, %f", 123, 'a', 3.14);
```

由于printf的函数参数并不是固定的，我们因此把这类函数称为可变参数函数。但在我们平时的编程当中，我们编写的函数都需要在函数头清晰地指出函数所需要的参数，也就是说，函数的参数在声明的时候就已经被固定下来了。那么printf是如何实现参数可变的呢？这就要用到C语言的可变参数机制。

C语言允许我们使用定义可变参数函数，此时函数的参数列表分为两部分，前一部分是固定下来的参数，如 `int`， `char *`， `double` 等用数据类型写出来的参数，后一部分是可变参数，其使用 `...` 来表示。例如，printf的声明如下。

```
int printf(char const* const _Format, ...);
```

printf的参数分为两部分，前一部分是格式化输出字符串，后一半是可变数量的输出的参数。

虽然我们可以方便的使用可变参数函数，但是我们在定义可变参数时需要遵循一些规则，如下所示。

1. 对于可变参数函数，参数列表中至少有一个固定参数。
2. 可变参数列表必须放在形参列表最后，也就是说，`...` 必须放在函数的参数列表的最后。

明白了这些规则之后，我们就定义一个可变参数 `function`，如下所示。

```
void function(int n, ...);
```

`function` 的参数分为两部分，`n` 是可变参数的数量，`...` 表示可变参数列表。在调用 `funcition` 时，我们会使用 `n` 个 `int` 参数来调用 `funtion`， `function` 将这 `n` 个 `int` 参数打印出来。

在实现 `function` 前，我们先来看如何在函数内部引用可变参数列表中的参数。

为了引用可变参数列表中的参数，我们需要用到 `<stdarg.h>` 头文件定义的一个变量类型 `va_list` 和三个宏 `va_start`， `va_arg`， `va_end`，这三个宏用于获取可变参数列表中的参数，用法如下。

宏	用法说明
`va_list`	定义一个指向可变参数列表的指针。
`void va_start(va_list ap, last_arg)`	初始化可变参数列表指针 `ap`，使其指向可变参数列表的起始位置，即函数的固定参数列表的最后一个参数 `last_arg` 的后面第一个参数。
`type va_arg(va_list ap, type)`	以类型 `type` 返回可变参数，并使 `ap` 指向下一个参数。
`void va_end(va_list ap)`	清零 `ap`。

为了引用可变参数列表的参数，我们也需要遵守一些规则，如下所示。

- 可变参数必须从头到尾逐个访问。如果你在访问了几个可变参数之后想半途中止，这是可以的，但是，如果你想一开始就访问参数列表中间的参数，那是不行的(可以把想访问的中间参数之前的参数读取但是不使用，曲线救国)。
- 这些宏是无法直接判断实际实际存在参数的数量。
- 这些宏无法判断每个参数的类型。所以在使用 `va_arg` 的时候一定要指定正确的类型。
- 如果在 `va_arg` 中指定了错误的类型，那么将会影响到后面的参数的读取。
- 第一个参数也未必要是可变参数个数，例如 `printf()`。

明白了上述规则后，我们现在来实现 `function`，实现如下。

```
1  void function(int n, ...)
2  {
3        // 定义一个指向可变参数的指针parameter
4        va_list parameter;
5        // 使用固定参数列表的最后一个参数来初始化parameter
6        // parameter指向可变参数列表的第一个参数
7        va_start(parameter, n);
8
9        for ( int i = 0; i < n; ++i ) {
10             // 引用parameter指向的int参数，并使parameter指向下一个参数
11             std::cout << va_arg(parameter, int) << " ";
12       }
13
14       // 清零parameter
15       va_end(parameter);
16
17       std::cout << std::endl;
18 }
```

第4行，我们首先定义一个指向可变参数列表的指针 `parameter`， `parameter` 会帮助我们引用可变参数列表的参数。但是，此时 `parameter` 并未指向 `function` 的可变参数列表，我们需要使用 `va_start` 来初始化 `parameter`，使其指向可变参数列表的第一个参数。为什么我们一定要指定一个固定参数呢？回想起第3章反复强调的C/C++函数调用规则——在函数调用前函数的参数会被从右到左依次入栈。也就是说，无论参数数量有多少，这些参数都被统一地放到了栈上，只不过可变参数函数并不知道这些栈上的参数具体含义。因此我们才需要使用 va_arg 来指定参数的类型后才能引用函数的可变参数。注意到栈的增长方式是从高地址向低地址增长的，因此函数的参数从左到右，地址依次增大。固定参数列表的最后一个参数的作用就是告诉我们可变参数列表的起始地址，如下所示。

$$\text{可变参数列表的起始地址} = \text{固定参数列表的最后一个参数的地址} + \text{这个参数的大小}$$

这就是 `va_start` 的工作。

初始化了 `parameter` 后，我们就使用 `parameter` 和 `va_arg` 来引用可变参数。可变参数的函数并不知道每一个可变参数的类型和具体含义，它只是在调用前把这些参数放到了栈上。而我们人为地在 `<stdarg.h>` 中定义了一些访问栈地址的宏，然后指定了这些参数的具体类型和使用这些宏来取出参数，这就是可变参数机制的实现思想。这也是为什么我们需要在 `va_arg` 中指明 `parameter` 指向的参数类型，因为只有函数实现者才知道函数想要的参数是什么。

从本质上来说， `parameter` 就是指向函数调用栈的一个指针，类似 `esp`，`ebp`， `va_arg` 按照指定的类型来返回 `parameter` 指向的内容。注意，在 `va_arg` 返回后， `parameter` 指向下一个参数，无需我们手动调整。

访问完可变参数后，我们需要使用 `va_end` 对 `parameter` 进行清零，这是防止后面再使用 `va_arg` 和 `parameter` 来引用栈中的内容，从而导致错误。

然后我们来使用这个函数，如下所示。

```
1   #include <iostream>
2   #include <stdarg.h>
3
4   void function(int n, ...);
5
6   int main()
7   {
8         function(1, 213);
9         function(2, 234, 2567);
10        function(3, 487, -12, 0);
11  }
12
13  void function(int n, ...)
14  {
15        // 定义一个指向可变参数的指针parameter
16        va_list parameter;
17        // 使用固定参数列表的最后一个参数来初始化parameter
18        // parameter指向可变参数列表的第一个参数
19        va_start(parameter, n);
```

```
20
21          for ( int i = 0; i < n; ++i ) {
22                // 引用parameter指向的int参数，并使parameter指向下一个参数
23                std::cout << va_arg(parameter, int) << " ";
24          }
25
26          // 清零parameter
27          va_end(parameter);
28
29          std::cout << std::endl;
30    }
```

编译运行后，输出如下结果。

```
1    213
2    234 2567
3    487 -12 0
```

学习完上面这个例子，我们对可变参数的机制已经有所了解，接下来我们就要实现可变参数的3个宏 `va_start`，`va_arg`，`va_end`。

可变参数机制的实现

可变参数并不神秘，从本质上来说，C语言首先提供我们定义可变参数列表需要的符号 `...`，有了这个符号后，我们任意地改变函数的调用参数时并不会发生错误，函数的参数都会被放到栈上面。此时，`<stdarg.h>` 提供了3个访问栈中的参数的宏。而这些可变参数的具体意义是什么，需要我们在使用这些宏的时候人为规定。这便是可变参数的本质，我们现在就来实现它。

首先，`va_list` 是指向可变参数列表的指针，其实就是字节类型的指针，而 `char` 类型就是1个字节，如下所示。

```
1    typdef char * va_list;
```

然后，我们定义 `va_start`，`va_start` 是初始化一个指向可变参数列表起始地址的指针 `ap`，需要用到固定参数列表的最后一个变量 `v`，如下所示。

```
1    #define _INTSIZEOF(n) ( (sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1) )
2    #define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )
```

由于栈中的push和pop的变量都是32位的，也就是4个字节。而ap是指向栈的，因此ap需要4个字节对齐，也就是4的倍数。`_INTSIZEOF(n)`返回的是n的大小进行4字节对齐的结果。注意到，4的倍数在二进制表示中的低2位是0，而任何地址和`0xfffffffc`(`~(sizeof(int)-1)`)相与后得到的数的低2位为0，也就是4的倍数。但是，直接拿一个数和`0xfffffffc`相与得到的结果是向下4字节对齐的，为了实现向上对齐，我们需要先加上`(sizeof(int)-1)`后再和`0xfffffffc`相与，此时得到的结果就是向上4字节对齐的。

`va_arg`的作用是返回`ap`指向的，`type`类型的变量，并同时使`ap`指向下一个参数，如下所示。

```
1    #define va_arg(ap, type) ( *(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
```

最后，`va_end`的作用是将`ap`清零，如下所示。

```
1    #define va_end(ap) ( ap = (va_list)0 )
```

至此，可变参数的机制已经实现完毕，结合可变参数机制的实现过程，同学们应该能够对可变参数的使用有了进一步的理解。

下面我们就使用我们实现的宏来引用可变参数，如下所示。

```
1   #include <iostream>
2
3   typedef char *va_list;
4   #define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
5   #define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
6   #define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
7   #define va_end(ap) (ap = (va_list)0)
8
9
10  void function(int n, ...);
11
12  int main()
13  {
14        function(1, 213);
15        function(2, 234, 2567);
16        function(3, 487, -12, 0);
17  }
18
19  void function(int n, ...)
20  {
21        // 定义一个指向可变参数的指针parameter
22        va_list parameter;
23        // 使用固定参数列表的最后一个参数来初始化parameter
24        // parameter指向可变参数列表的第一个参数
```

```
25          va_start(parameter, n);
26
27          for (int i = 0; i < n; ++i)
28          {
29                  // 引用parameter指向的int参数，并使parameter指向下一个参数
30                  std::cout << va_arg(parameter, int) << " ";
31          }
32
33          // 清零parameter
34          va_end(parameter);
35
36          std::cout << std::endl;
37  }
```

编译运行后，得到相同的结果。

```
1   213
2   234 2567
3   487 -12 0
```

此时，我们并未引入头文件 `<stdarg.h>`，这说明了我们已经成功实现了可变参数机制。

实现printf

学会了可变参数后，printf的实现便不再困难。在实现printf前，我们首先要明白printf的作用。printf的作用是格式化输出，并返回输出的字符个数，其定义如下。

```
1   int printf(const char *const fmt, ...);
```

在格式化输出字符串中，会包含 `%c,%d,%x,%s` 等来实现格式化输出，对应的参数在可变参数中可以找到。明白了printf的作用，printf的实现便迎刃而解，实现思路如下。

printf首先找到fmt中的形如 `%c,%d,%x,%s` 对应的参数，然后用这些参数具体的值来替换 `%c,%d,%x,%s` 等，得到一个新的格式化输出字符串，称为fmt的解析，最后将这个新的格式化输出字符串即可。但是，这个字符串可能非常大，会超过函数调用栈的大小。因此我们实际上会定义一个缓冲区，然后对fmt进行逐字符地解析，将结果逐字符的放到缓冲区中。放入一个字符后，我们会检查缓冲区，如果缓冲区已满，则将其输出，然后清空缓冲区，否则不做处理。

在实现printf前，我们需要一个能够输出字符串的函数，这个函数能够正确处理字符串中的 `\n` 换行字符。这里，有同学会产生疑问，`\n` 不是直接输出就可以了吗？其实 `\n` 的换行效果是我们人为规定的，换行的实现需要我们把光标放到下一行的起始位置，如果光标超过了屏幕的表示范围，则需要滚屏。因此，我们实现一个能够输出字符串的函数 `STDIO::print`，声明和实现分别放在 `include/stdio.h` 和 `src/io/stdio.cpp` 中，如下所示。

```
int STDIO::print(const char *const str)
{
	int i = 0;

	for (i = 0; str[i]; ++i)
	{
		switch (str[i])
		{
		case '\n':
			uint row;
			row = getCursor() / 80;
			if (row == 24)
			{
				rollUp();
			}
			else
			{
				++row;
			}
			moveCursor(row * 80);
			break;

		default:
			print(str[i]);
			break;
		}
	}

	return i;
}
```

> 在程序设计中，命名是一件令人苦恼的事情，而借助于C++的函数重载，我们可以将许多功能类似的函数用统一的名字来表示。

我们实现的printf比较简单，只能解析如下参数。

符号	含义
%d	按十进制整数输出
%c	输出一个字符
%s	输出一个字符串
%x	按16进制输出

printf的实现如下。

```
int printf(const char *const fmt, ...)
{
      const int BUF_LEN = 32;

      char buffer[BUF_LEN + 1];
      char number[33];

      int idx, counter;
      va_list ap;

      va_start(ap, fmt);
      idx = 0;
      counter = 0;

      for (int i = 0; fmt[i]; ++i)
      {
            if (fmt[i] != '%')
            {
                  counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
            }
            else
            {
                  i++;
                  if (fmt[i] == '\0')
                  {
                        break;
                  }

                  switch (fmt[i])
                  {
                  case '%':
                        counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
                        break;

                  case 'c':
                        counter += printf_add_to_buffer(buffer, va_arg(ap, int), idx, BUF_LEN);
                        break;

                  case 's':
                        buffer[idx] = '\0';
                        idx = 0;
                        counter += stdio.print(buffer);
                        counter += stdio.print(va_arg(ap, const char *));
                        break;

                  case 'd':
```

```
                case 'x':
                    int temp = va_arg(ap, int);

                    if (temp < 0)
                    {
                        counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
                        temp = -temp;
                    }

                    temp = itos(number, temp, (fmt[i] == 'd' ? 10 : 16));

                    for (int j = temp - 1; j >= 0; --j)
                    {
                        counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
                    }
                    break;

                }
            }
        }

        buffer[idx] = '\0';
        counter += stdio.print(buffer);

        return counter;
}
```

首先我们定义一个大小为 `BUF_LEN` 的缓冲区 `buffer`， `buffer` 多出来的1个字符是用来放置 `\0` 的。由于我们后面会将一个整数转化为字符串表示， `number` 使用来存放转换后的字符串的。

接着我们开始对 `fmt` 进行逐字符解析，对于每一个字符 `fmt[i]`，如果 `fmt[i]` 不是 `%`，则说明是普通字符，直接放到缓冲区即可。注意，将 `fmt[i]` 放到缓冲区后可能会使缓冲区变满，此时如果缓冲区满，则将缓冲区输出并清空，上述过程写成一个函数来实现，如下所示。

```
int printf_add_to_buffer(char *buffer, char c, int &idx, const int BUF_LEN)
{
    int counter = 0;

    buffer[idx] = c;
    ++idx;

    if (idx == BUF_LEN)
    {
        buffer[idx] = '\0';
        counter = stdio.print(buffer);
        idx = 0;
    }
```

```
14
15        return counter;
16    }
```

如果 `fmt[i]` 是 `%`，则说明这可能是一个格式化输出的参数。因此我们检查 `%` 后面的参数，分为如下情况分别处理。

- `%%` 。输出一个 `%` 。
- `%c` 。输出 `ap` 指向的字符。
- `%s` 。输出 `ap` 指向的字符串的地址对应的字符串。
- `%d` 。输出 `ap` 指向的数字对应的十进制表示。
- `%x` 。输出 `ap` 指向的数字对应的16进制表示。
- 其他。不做任何处理。

一个数字向任意进制表示的字符串的转换函数如下所示，声明放置在 `include/stdlib.h` 中，实现放置在 `src/utils/stdlib.cpp` 中。

```
1   int itos(char *numStr, int num, int mod) {
2         if (mod < 2 || mod > 26 || num < 0) {
3               return -1;
4         }
5
6         int length, temp;
7
8         length = 0;
9         while(num) {
10              temp = num % mod;
11              num /= mod;
12              numStr[length] = temp > 9 ? temp - 10 + 'A' : temp + '0';
13              ++length;
14        }
15
16        if(!length) {
17              numStr[0] = '0';
18              ++length;
19        }
20
21        return length;
22  }
```

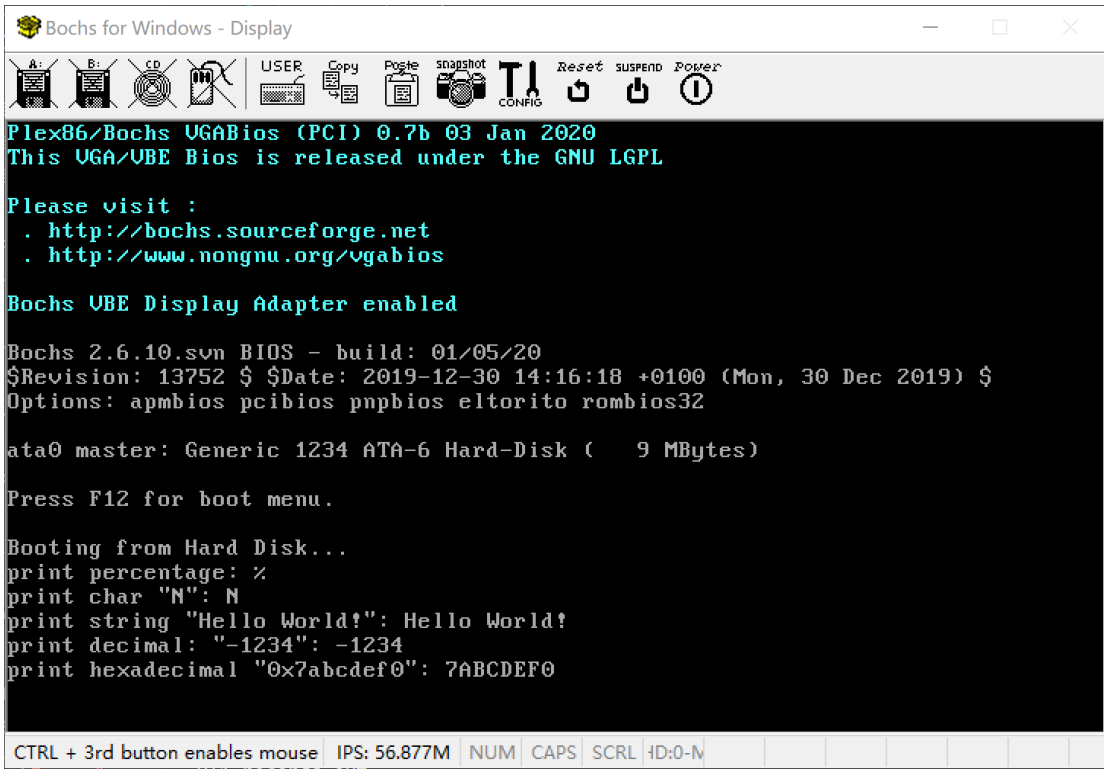
最后，当我们逐字符解析完 `fmt` 后， `buffer` 中可能还会有未输出的字符，我们要将其输出，然后返回输出的总字符 `counter` 。

接下来我们测试这个函数，我们在 `setup_kernel` 中加入对应的测试语句。

```
```

```cpp
#include "os_type.h"
#include "os_modules.h"
#include "asm_utils.h"
#include "stdio.h"
#include "stdlib.h"

extern "C" void setup_kernel()
{
        // 中断处理部件
        interruptManager.initialize();
        // 屏幕IO处理部件
        stdio.initialize();
        interruptManager.enableTimeInterrupt();
        interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
        //asm_enable_interrupt();
        printf("print percentage: %%\n"
                "print char \"N\": %c\n"
                "print string \"Hello World!\": %s\n"
                "print decimal: \"-1234\": %d\n"
                "print hexadecimal \"0x7abcdef0\": %x\n",
                'N', "Hello World!", -1234, 0x7abcdef0);
        //uint a = 1 / 0;
        asm_halt();
}
```

然后编译运行，输出如下结果。



至此，我们实现了一个简单的 `printf` 函数。

> 此后，我们就可以愉快地使用printf来debug啦~

二级分页机制

缘由

虽然计算机的内存容量在不断增大，容量也从原先的B级发展为今天的GB级，但是，我们依然无法将所有的程序放入内存中执行。因此，我们要确定哪些程序是可以放入内存的，哪些程序是暂时放在外存。除此之外，当我们实现了多线程和多进程后，我们希望这些线程和进程在运行时互不干扰，即它们无法访问对方的地址空间，从而实现了内存保护。

上面实际上提到了两个问题。

- 程序的装入。
- 内存的保护。

我们首先来看程序的装入。在第3章中我们已经了解到一份C代码需要经过编译、链接后才能形成可执行程序。但是，这个可执行程序是放在外存中的，也就是磁盘，它需要被加载到内存中才能被执行，程序从外存加载到内存的执行过程被称为程序的装入。

特别注意，程序在被编译后，其起始地址默认是从0开始的。但是，在绝大部份情况下，我们的程序并不是加载到地址为0处运行。例如，我们的bootloader是被加载到 `0x7e00` 处执行的。如果我们直接将编译后的bootloader加载到 `0x7e00` 处执行就会导致我们无法正确的访问代码中定义的变量，这就是为什么我们会出现如下的语句。

```
1    ;以下进入保护模式
2    jmp dword CODE_SELECTOR:protect_mode_begin + LOADER_START_ADDRESS
```

因为标号 `protect_mode_begin` 表示的是相对于代码起始位置的偏移地址，假设这个偏移地址是 `0x200` ，这条语句是希望跳转到标号为 `protect_mode_begin` 的代码处执行。但是，由于程序在编译后的起始地址默认是从0开始的，因此如果使用下面的语句

```
1    ;以下进入保护模式
2    jmp dword CODE_SELECTOR:protect_mode_begin
```

我们实际上会跳转到 `0x200` 处执行(`CODE_SELECTOR` 对应的段描述符的基地址从0开始)，但是，由于bootloader是被加载到 `0x7e00` 的，我们实际上需要跳转到 `0x7e00+0x200` 处才是 `protect_mode_begin` 的代码所在。此时，我们需要手动为标号 `protect_mode_begin` 加上一个bootloader被加载的起始地址 `LOADER_START_ADDRESS` 后才能正确执行，这也是第1章的bonus-2的答案。

对于C语言也是如此，我们的 `kernel` 是使用C语言写的，C语言的全局变量(如 `stdio` ，函数名等)的地址也都是偏移地址，而kernel是被加载到 `0x20000` 处执行的。但奇怪的是，但我们并未给这些全局变量加上一个表示kernel加载的位置的变量，此时也能够正确访问。这是因为我们在使用 `ld` 链接的时候，加上了 `-Ttext 0x00020000` 参数，此时 `ld` 会修改程序中所有的全局变量的地址，为其加上 `0x20000` ，这就是运行不会出错的原因。上面这个修改编译后的程序的地址然后再装入内存执行的过程被称为程序的装入，而程序在编译后的起始地址为0也是为程序的装入服务的。

> 对于局部变量，C语言翻译成的汇编代码是使用ebp和esp来访问的，因此不会出错。

此时，为了区分程序编译后的地址和程序运行时的地址，我们就有了线性地址和物理地址的概念。线性地址是程序编译后的地址，默认从0开始；物理地址是程序运行时的地址，是程序被加载到内存的地址，也是CPU寻址所用的地址。线性地址需要经过转换后才能变成CPU寻址所用的物理地址，线性地址转换成物理地址的过程是程序的装入的一部分，根据转换方式的不同，我们可以将程序的装入划分为3种方式。

- 绝对装入。在链接时，若知道程序将驻留在内存的某个地址，则链接程序将根据实际运行的地址来修改程序的线性地址，使其和物理地址保持一致。例如 `ld` 中的 `-Ttext 0x00020000` ，此时直接将程序加载到预先确定的位置便可运行。但是，当加载位置变化后，链接时的地址也要发生变化，否则必定发生错误。
- 静态重定位。在装入时，我们根据程序被加载位置来修改程序的指令和数据地址。这和绝对装入类似，区别在于对于绝对装入程序的线性地址就是实际的物理地址，而对于静态重定位，程序的线性地址依旧是从0开始，只有在被加载到内存时才会被修改。
- 动态重定位。程序被装入内存后依旧是线性地址，线性地址的转换被推迟到程序需要寻址时才进行。此时，我们需要一个内存管理单元(Memory Management Unit, MMU)来帮助我们将线性地址翻译成CPU寻址所用的物理地址。我们可以改变MMU的内容来实现不同的变换方式，也就是说，即使对于相同的线性地址，在不同的变换方式下，得到的物理地址就会不同，这就是被称为动态重定位的原因。动态重定位的代表就是我们接下来要实现的二级分页机制。

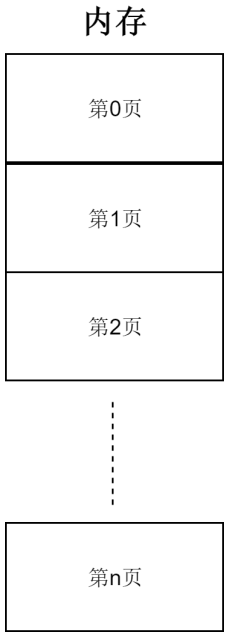
二级分页机制的引入有效地帮助我们来进行程序的动态重定位，在下面我们可以看到，我们可以把连续的线性地址通过MMU映射到不连续的物理地址处，大大提高了内存的利用效率。除此之外，对于不同的程序，即使它们的线性地址都是从0开始的，但是在不同的地址变换方式下，最终得到了不同的物理地址，从而实现了内存地址的保护。

我们接下来首先看一下什么是分页机制。

一级页表

> 线性地址也会被称作虚拟地址。

在分页机制下，内存被划分为大小相等的内存块，称为页（Page），如下所示。



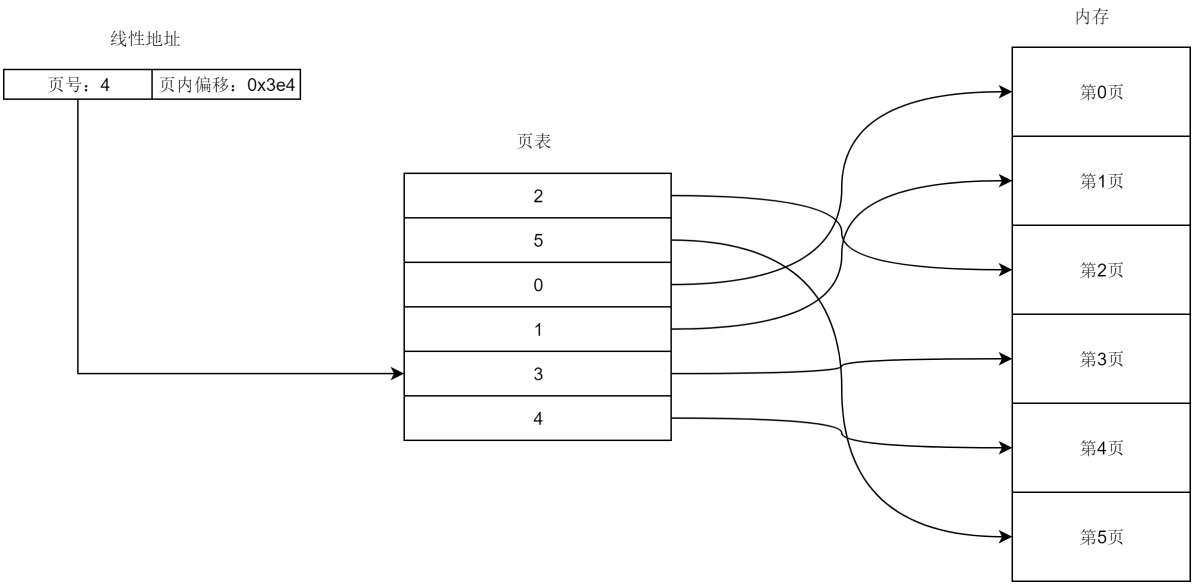
在现代CPU中，页的大小为4KB。我们知道，CPU最终会使用物理地址来访问程序的代码和数据。在未开启分页机制前，物理地址=段地址+偏移地址。而开启了分页机制后，分段机制依然在发挥作用，只不过段地址+偏移地址不再是实际的物理地址，而是被称为线性地址。线性地址需要经过转换才能变为物理地址，负责这一转换过程的部件称为MMU。MMU集成在CPU中，程序的运行过程虽然给出的是线性地址，但是CPU能够通过页部件将其自动地转换为物理地址。

一级页表先通过线性地址在页表中找到对应的页表项，然后再根据页表项给出的页地址和页内偏移找到实际的物理地址。页的大小是4KB，用低12位来表示（$4K = 2^{12}$）。我们知道，保护模式下的地址是4B，32位表示。地址的高20位用来确定页表中的页表项，所以页表中的页表项一共有$2^{20} = 1M$项。每一个页表项占4字节，里面保存的是页的物理地址。因此，1M个页表项恰好可以表示$1M * 4KB = 4GB$的内存，也就是32位的地址空间。

综合上述分析，一级页表的线性地址到物理地址的转换关系如下。

先取线性地址的高20位，高20位的数值表示的是页表项在页表中的序号。而每一个页表项占4字节，所以高20位的数值乘4后才是对应的页表项的地址。从页表项中读出页地址后，由于低20位是页内偏移，使用页地址+低20位即可得出需要访问的物理地址。

我们看下面这个例子。考虑线性地址 `0x43e4` 。



线性地址到物理地址的变换过程如下所示。

线性地址的高20位是 `0x4` ，因此我们到页表中去找第 `0x4` 个页表项，页表项的内容为3，表示线性地址对应的物理页是第3个物理页。线性地址的第12位表示的是页内偏移，因此线性地址 `0x43e4` 对应的地址是第3个物理页中偏移地址为 `0x3e4` 的地址。由于一个物理页的大小为4KB，线性地址对应的物理地址是 `0x33e4` ，实际上就是用物理页号去替换线性地址中的页号。

明白了一级分页机制的基本思想后，我们便不难扩展到二级页表分页机制。

二级页表

我们已经看到，一级页表已经可以实现分页机制的思想。但是，问题在于一级页表中最多可容纳1M个页表项，每个页表项是4字节，如果页表项全满的话，便是4MB大小。每个进程都有自己的页表，进程一多，光是页表占用的空间就很可观了。 所以，我们不希望在一开始就创建所有的页表项，而是根据需要动态地创建页表项。所以借用分页机制的思想，我们也为页表创建页表，称为页目录表，页目录表中的每一项被称为页目录项，页目录项的内容是对应页表的物理地址。通过页目录表来访问页表，然后通过页表访问物理页的方式被称为二级页表机制。

在一级页表机制下，页表的大小为4MB，而在二级页表机制下，页目录表、页表、物理页的大小均为4KB。页目录项和页表项的大小均为4B。但是，这种划分方式是否足以表示4GB的内存空间呢？我们接下来分析。页目录表中的页目录项为$4KB/4B = 1024$，每一个页目录项对应了一个页表，因此我们有1024个页表。每一个页表种的页表项的数目为$4KB/4B = 1024$，每一个页表项的对应一个物理页。因此1024个页表会对应$1024 \times 1024 = 1M$的物理页。每一个物理页的大小为4KB，所以二级页表可表示的总内存大小为$1M * 4KB = 4GB$，恰好为32位的地址空间。

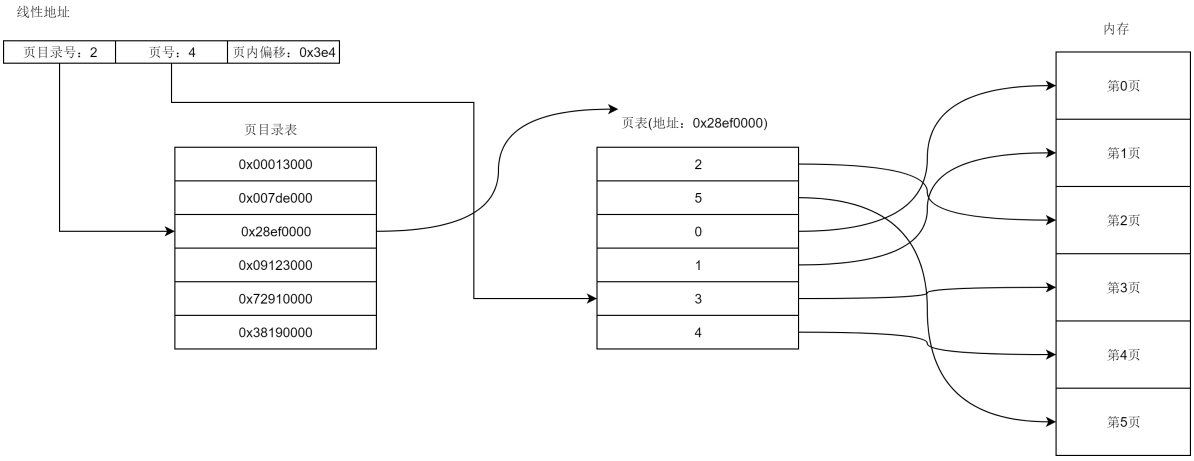
此时，32位地址被划分为3部分。

- 31-22，共10位，是页目录项的序号，恰好表示$2^{10} = 1024$个页目录项。
- 21-12，共10位，是页表项的序号，恰好表示$2^{10} = 1024$个页表项。
- 11-0，共12位，是页内偏移，恰好表示$2^{12} = 4KB$的物理页。

综合上述分析，二级页表的线性地址到物理地址的转换关系如下。

1. 给定一个线性地址，先取31-22位，其数值乘4B后得到页目录表项在页目录表的偏移地址，加上页目录表的物理地址后得到页目录项的物理地址。
2. 取页目录项中的内容，即页表的物理地址，加上21-12位乘4B的结果后找到对应的页表项的物理地址。
3. 取页表项的内容，即物理页的物理地址，加上11-0位的内容后便得到实际的物理地址。

我们通过一个例子来说明，考察线性地址 `0x008043e4` 。



1. 取31-22位，数值为2，因此在页目录表中找到第2个页目录项。
2. 取第2个页目录项的内容，得到页表的地址0x28ef0000。取21-12位，数值为4，因此在页表中找到第4个页表项。
3. 取第4个页表项的内容，得到物理页是第3页。取11-0位，数值为 `0x3e4` ，因此在第3页中找到偏移地址为 `0x3e4` 处的值。

线性地址 `0x008043e4` 被转换成物理地址 `0x000033e4` ，这个过程是CPU中的地址转换部件自动完成的，我们只需要提供地址转换需要用到的页目录表和页表即可。

这便是二级页表的分页机制，我们接下来就开启它。

开启二级页表分页机制

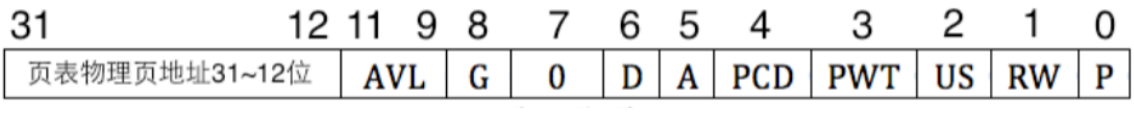
IA-32的CPU支持的是二级页表，所以我下面所说的分页机制指的是二级页表分页机制。二级页表分页机制默认是关闭的，我们需要手动开启，启动分页机制的流程如下所示。

- 规划好页目录表和页表在内存中的位置并写入内容。
- 将页目录表的地址写入cr3。
- 将cr0的PG位置1。

我们分别来看上面的步骤。

规划好页目录表和页表在内存中的位置并写入内容。首先，我们需要明确的一点是，页目录表和页表是需要在内存中特意得划分出位置来存放的。所以，我们需要规定页目录表和页表的位置，而且页目录表和页表的物理地址必须是4KB的整数倍。

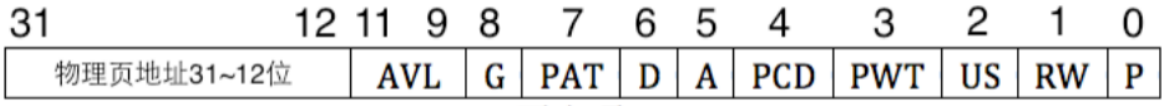
规定了页目录表的位置后，我们根据我们需要使用的线性地址来确定需要分配的页表的数量和位置，不必一开始就分配完1024个页表给页目录表。规划好了页表后，首先向页目录表中写入页表对应的页目录项。页目录项的结构如下。



- 31-12位是页表的物理地址位的高20位，这也是为什么规定了页目录表的地址必须是4KB的整数倍。页目录表和页表实际上也是内存中的一个页，而内存被划分成了大小为4KB的页，自然地这些物理页的起始就是4KB的整数倍。
- P位是存在位，1表示存在，0表示不存在。

- RW位，read/write。1表示可读写，0表示可读不可写。
- US位，user/supervisor。若为1时，表示处于User级，任意级别（ 0、 1、 2、3）特权的程序都可以访问该页。若为0，表示处于 Supervisor 级，特权级别为3的程序不允许访问该页，该页只允许特权级别为0、1、2的程序可以访问。
- PWT位，这里置0。PWT， Page-level Write-Through，意为页级通写位，也称页级写透位。若为 1 表示此项采用通写方式，表示该页不仅是普通内存，还是高速缓存。
- PCD位，这里置0。PCD， Page-level Cache Disable，意为页级高速缓存禁止位。若为 1 表示该页启用高速缓存，为 0 表示禁止将该页缓存。
- A位，访问位。1表示被访问过，0表示未被访问，由CPU自动置位。
- D位，Dirty，意为脏页位。当CPU对一个页面执行写操作时，就会设置对应页表项的D位为1。此项
  仅针对页表项有效，并不会修改页目录项中的D位。
- G位，这里置0，和TLB相关。
- PAT， 这里置0。Page Attribute Table，意为页属性表位，能够在页面一级的粒度上设置内存属性。

写完页目录项后便写页表项，页表项结构如下。



除31-12位是页的物理地址位的高20位外，其余位的意义与页目录项完全相同，这里便不再赘述。

将页目录表的地址写入cr3。cr3寄存器保存的是页目录表的地址，使得CPU的MMU能够找到页目录表的地址，然后自动地将线性地址转换成物理地址。我们在建立页目录表和页表后，需要将页目录表地址放到CPU所约定的地方，而这个地方是cr3。cr3又被称为页目录基址寄存器 PDBR，其内容如下。

cr3可以直接使用mov指令赋值。

将cr0的PG位置1。启动分页机制的开关是将控制寄存器 cr0 的 PG 位置 1，PG 位是cr0寄存器的第31位，PG位为1后便进入了内存分页运行机制。

综合上述3步骤后便可以开启分页机制，我们下面使用代码实现。

我们先创建内存管理器 `MemoryManager` ，然后加入开启分页机制的成员函数声明，代码放在 `include/memeory.h` 中，如下所示。

```
#ifndef MEMORY_H
#define MEMORY_H

class MemoryManager
{
public:
	void openPageMechanism();
};
#endif
```

同样地，我们在 `include/modules` 加入内存管理器的定义。

```
#ifndef OS_MODULES_H
#define OS_MODULES_H

#include "interrupt.h"
#include "stdio.h"
#include "memory.h"

// 中断管理器
InterruptManager interruptManager;
// 输出管理器
STDIO stdio;
// 内存管理器
MemoryManager memoryManager;

#endif
```

然后我们在 `src/memory/memory.cpp` 中实现这个函数，如下所示。

```
void MemoryManager::openPageMechanism()
{
	// 页目录表指针
	int *directory = (int *)PAGE_DIRECTORY;
	//线性地址0~4MB对应的页表
	int *page = (int *)(PAGE_DIRECTORY + PAGE_SIZE);

	// 初始化页目录表
	memset(directory, 0, PAGE_SIZE);
	// 初始化线性地址0~4MB对应的页表
	memset(page, 0, PAGE_SIZE);

	int address = 0;
	// 将线性地址0~1MB恒等映射到物理地址0~1MB
	for (int i = 0; i < 256; ++i)
```

```
16      {
17              // U/S = 1, R/W = 1, P = 1
18              page[i] = address | 0x7;
19              address += PAGE_SIZE;
20      }
21
22      // 初始化页目录项
23
24      // 0~1MB
25      directory[0] = ((int)page) | 0x07;
26      // 3GB的内核空间
27      directory[768] = directory[0];
28      // 最后一个页目录项指向页目录表
29      directory[1023] = ((int)directory) | 0x7;
30
31      // 初始化cr3，cr0，开启分页机制
32      asm_init_page_reg(directory);
33
34      printf("open page mechanism\n");
35
36  }
```

其中，常量的定义如下所示。

```
1   #define PAGE_SIZE 4096
2   #define PAGE_DIRECTORY 0x100000
```

我们现在来分析上面这段代码。

我们打算将页目录表放在 `1MB` 处。在开启分页机制前，我们需要建立好内核所在地址的页目录表和页表，否则一旦置PG位=1，开启分页机制后，CPU就会出现寻址异常。由于我们的内核很小，可以放在0~1MB的内存区域。为了访问方便，对于0~1MB的内存区域我们建立的是线性地址到物理地址的恒等映射，也就是说，线性地址就是物理地址。这个时候，我们就要设置相应的页目录项和页表项。

> 在开启分页机制后，同学们遇到一个线性地址时，切记将其拆开三部分，根据页目录号和页表号找到的物理页地址加上偏移地址后才是真正的物理地址，这一点在刚开始接触时并不能非常直观地理解。

首先，0~1MB的线性地址范围是0x00000000~0x000fffff，其31-22位均为0，对应第0个页目录项。因此我们只需要一个页表，这个页表被放在页目录表之后，地址是 `PAGE_DIRECTORY + PAGE_SIZE` 。然后我们取21~12位，范围从0x000~0xfff，涉及256个页表项。由于我们希望线性地址经过翻译后的物理地址依然和线性地址相同，因此，这256个页表项分别对应物理页的第0页，第1页和第255页，第15~20行用于设置这256页表项，如下所示。

```
        for (int i = 0; i < 256; ++i)
        {
                // U/S = 1, R/W = 1, P = 1
                page[i] = address | 0x7;
                address += PAGE_SIZE;
        }
```

除了设置页表项对应的物理页地址和固定为0的位外，我们设置U/S，R/W和P位为1。

然后我们初始化页目录项，由于我们的0~1MB的线性地址对应于第0个页目录项，我们用刚刚放入了256个页表项的页表作为第0个页目录项指向的页表。同样地，我们设置U/S，R/W和P位为1。

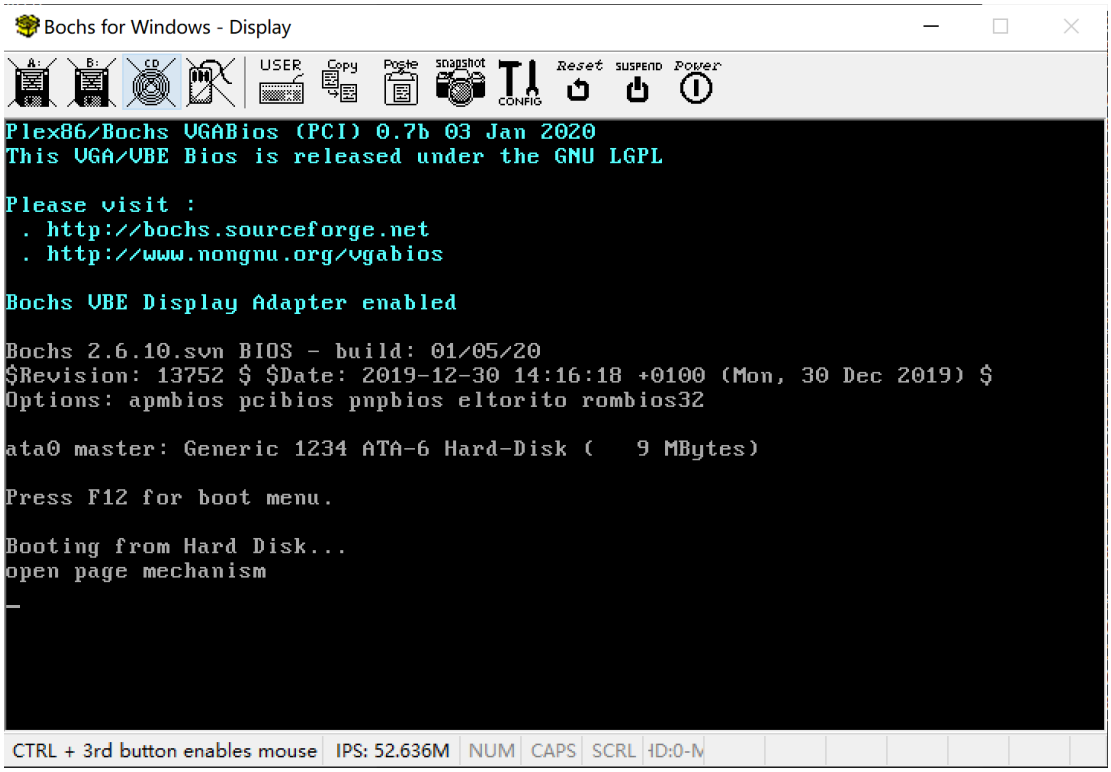
后面我们设置第768个页目录项和第0个页表项相同、设置最后一个页目录项指向页目录表，这是为了后面实现用户进程服务的，这里按下不表。

然后我们将页目录表的地址放入cr3寄存器，将cr0的PG位置1便可开启分页机制，如下所示。

```
asm_init_page_reg:
        push ebp
        mov ebp, esp

        push eax

        mov eax, [ebp + 4 * 2]
        mov cr3, eax ; 放入页目录表地址
        mov eax, cr0
        or eax, 0x80000000
        mov cr0, eax                ; 置PG=1，开启分页机制

        pop eax
        pop ebp

        ret
```

置PG=1后，开启分页机制，函数如果能正确返回，则说明CPU能够正确使用分页机制来寻址。

我们编译项目，然后使用bochs加载运行，结果如下。



此时，程序能够正确返回并输出相应的语句，说明分页机制开启成功。使用 `info tab` 可以查看页表的内容，如下所示。

```
<bochs:2> info tab
cr3: 0x000000100000
0x0000000000000000-0x00000000000fffff -> 0x000000000000-0x0000000fffff
0x00000000c0000000-0x00000000c00fffff -> 0x000000000000-0x0000000fffff
0x00000000ffc00000-0x00000000ffc00fff -> 0x000000101000-0x000000101fff
0x00000000fff00000-0x00000000fff00fff -> 0x000000101000-0x000000101fff
0x00000000fffff000-0x00000000ffffffff -> 0x000000100000-0x000000100fff
```

至此，本章的内容已经讲述完毕。

> 本章的内容虽然不多，但是理解起来抽象，同学们注意仔细体会😅

练习

1. 请根据 `va_list`， `va_start`， `va_arg`， `va_end` 的宏定义来分析其实现思路。
2. 请结合资料说说C语言的printf是如何实现的。
3. 扩展printf的实现。
4. 请问回车和换行有什么区别， `\n` 的效果是回车还是换行。
5. 请完成第一章的bonus 2。
6. 请谈谈段式，页式，段页式三种内存管理机制的区别。
7. 请分别在段式，页式，段页式三种内存管理机制下谈谈线性地址和物理地址的区别与联系。
8. 使用一级页表的图，分析线性地址 `0x2918` 是如何被转换成物理地址的。

9. 使用二级页表的图，分析线性地址 `0x802918` 是如何被转换成物理地址的。
10. 请分析如何为线性地址 `0x23492340~0x2359ffff` 建立页目录表和页表。

bonus

1. 在开启分页机制后，我们使用了指令 `info tab` 输出了如下内容，请分析输出的内容的含义，并简要分析背后的原因。(Tips：注意我们只向页目录表放入了3个页目录项，只在一个页表中放入了256个页表项，输出的内容和这些页目录项、页表项和线性地址向物理地址的转换过程相关)

```
<bochs:2> info tab
cr3: 0x000000100000
0x0000000000000000-0x00000000000fffff -> 0x000000000000-0x0000000fffff
0x00000000c0000000-0x00000000c00fffff -> 0x000000000000-0x0000000fffff
0x00000000ffc00000-0x00000000ffc00fff -> 0x000000101000-0x000000101fff
0x00000000fff00000-0x00000000fff00fff -> 0x000000101000-0x000000101fff
0x00000000fffff000-0x00000000ffffffff -> 0x000000100000-0x000000100fff
```