第5章 内核线程

> what you see is not what you get.

写在前面

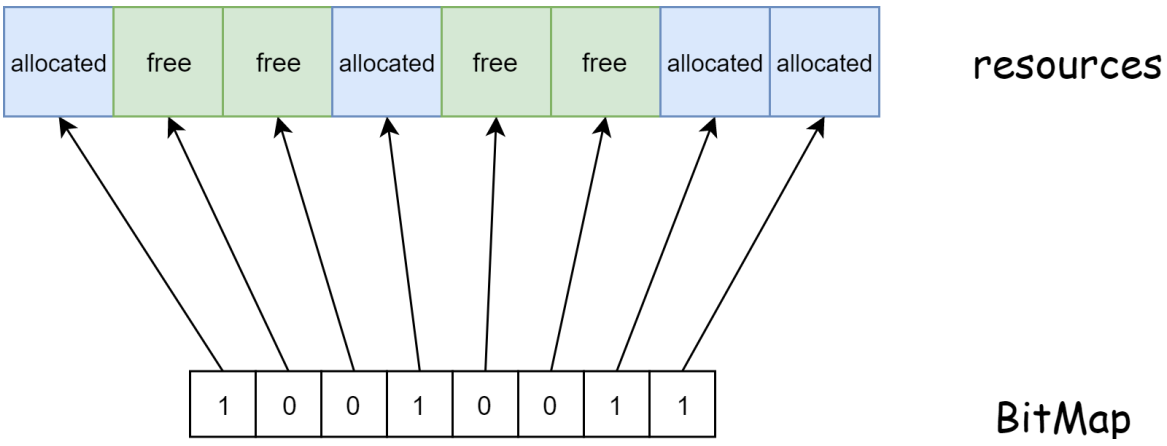
在本章中，同学们会了解到如下知识。

- 学习使用位图来将进行资源管理。
- 实现页内存分配。
- 学习内核线程的创建、调度和回收。

页内存分配

位图

位图，BitMap，是用于资源管理的一种数据结构。BitMap使用1位来和资源单元建立映射关系，从而以较小的代价实现对资源的管理，如下所示。



在上面的例子中，我们使用1来表示资源单元已经被分配，0表示资源单元空闲的。

对于4GB的内存，以分页机制管理，则资源单元为一个物理页，大小为4KB。因此，使用BitMap来对该内存进行管理时，BitMap的大小为

$$\frac{4GB}{8*4KB}=128KB.$$

其内存占比为

$$\frac{128KB}{4GB}*100\%=0.00305\%.$$

可见使用BitMap来对资源管理时，其空间效率较高。

Assignment 1 实现bitmap

从代码层面上去考察BitMap时，我们需要为BitMap提供一块存放BitMap的内存区域和BitMap管理的资源单元数量，BitMap则提供单独存取位和批处理存取位的方法，如下所示，BitMap的定义放置在 `include/bitmap.h` 中。

```
1   class BitMap
2   {
3   public:
4         // 被管理的资源个数，bitmap的总位数
5         int length;
6         // bitmap的起始地char址
7         char *bitmap;
8   public:
9         // 初始化
10        BitMap();
11        // 设置BitMap，bitmap=起始地址，length=总位数(被管理的资源个数)
12        void initialize(char *bitmap, const int length);
13        // 获取第index个资源的状态，true=allocated，false=free
14        bool get(const int index) const;
15        // 设置第index个资源的状态，true=allocated，false=free
```

```
        void set(const int index, const bool status);
        // 分配count个连续的资源，若没有则返回-1，否则返回分配的第1个资源单元序号
        int allocate(const int count);
        // 释放第index个资源开始的count个资源
        void release(const int index, const int count);
        // 返回Bitmap存储区域
        char *getBitmap();
        // 返回Bitmap的大小
        int size() const;
private:
        // 禁止Bitmap之间的赋值
        BitMap(const BitMap &) {}
        void operator=(const BitMap&) {}
};
```

注意，BitMap的成员是有指针的。一般情况下，成员涉及指针的对象的赋值都需要使用动态内存分配获得一个新的指针，但为了避免不必要的错误，我们将 `copy constructor` 和 `operator=` 定义为private来禁止BitMap之间的直接赋值。这也是为什么我们在BitMap的初始化函数 `initialize` 中需要提供bitmap的存储区域。

根据BitMap的描述，我们不难实现BitMap的成员函数，如下所示，代码放置在 `src/utils/bitmap.cpp` 中。

```
BitMap::BitMap()
{
}

void BitMap::initialize(char *bitmap, const int length)
{
        this->bitmap = bitmap;
        this->length = length;

        int bytes = ceil(length, 8);

        for (int i = 0; i < bytes; ++i)
       {
              bitmap[i] = 0;
       }
}

bool BitMap::get(const int index) const
{
        int pos = index / 8;
        int offset = index % 8;

        return (bitmap[pos] & (1 << offset));
}
```

```
25
26  void BitMap::set(const int index, const bool status)
27  {
28        int pos = index / 8;
29        int offset = index % 8;
30
31        // 清0
32        bitmap[pos] = bitmap[pos] & (~(1 << offset));
33
34        // 置1
35        if (status)
36        {
37              bitmap[pos] = bitmap[pos] | (1 << offset);
38        }
39  }
40
41  int BitMap::allocate(const int count)
42  {
43        if (count == 0)
44              return -1;
45
46        int index, empty, start;
47
48        index = 0;
49        while (index < length)
50        {
51              // 越过已经分配的资源
52              while (index < length && get(index))
53                    ++index;
54
55              // 不存在连续的count个资源
56              if (index == length)
57                    return -1;
58
59              // 找到1个未分配的资源
60              // 检查是否存在从index开始的连续count个资源
61              empty = 0;
62              start = index;
63              while ((index < length) && (!get(index)) && (empty < count))
64              {
65                    ++empty;
66                    ++index;
67              }
68
69              // 存在连续的count个资源
70              if (empty == count)
71              {
72                    for (int i = 0; i < count; ++i)
73                    {
```

```
                        set(start + i, true);
                }

                return start;
            }
        }

        return -1;
}

void BitMap::release(const int index, const int count)
{
        for (int i = 0; i < count; ++i)
        {
                set(index + i, false);
        }
}

char *BitMap::getBitmap()
{
        return (char *)bitmap;
}

int BitMap::size() const
{
        return length;
}
```

上面的代码逻辑比较清晰，因此我只说明如何找到资源单元对应的状态位。

我们使用指针来访问bitmap的存储区域时，最小的访问单元是字节，而资源单元的状态是使用一个位来表示的。因此，给定一个资源单元的序号i，我们无法通过`bitmap[i]`的方式来直接修改资源单元的状态。我们的做法是先定位到存储第i个资源单元的字节序号pos，然后在确定第i个资源单元的状态位在第pos字节中的偏移位置$offset$，最后使用位运算来修改该位即可，如下所示。

$$i = 8 \cdot pos + offset,\ 0 \leq offset < 8$$

注意，$offset$是从最低位开始算起的。

地址池

为了实现页内存分配，我们需要使用一种结构来标识地址空间中的哪些页是已经被分配的，哪些是未被分配的。这种结构被称为地址池，当需要页内存分配时，我们可以从地址池中取出一个空闲页。然后地址池便会标识该内存块已被分配，只要该内存块没有被释放，否则不会被再次分配。注意到BitMap可用于资源的管理，所以地址池实际上使用了BitMap来对其地址空间进行管理。

从代码层面上来看，地址池的定义如下所示，代码放置在`include/address_pool.h`中。

```
class AddressPool
{
public:
	BitMap resources;
	int startAddress;
public:
	AddressPool();
	// 初始化地址池
	void initialize(char *bitmap, const int length,const int startAddress);
	// 从地址池中分配count个连续页，成功则返回第一个页的地址，失败则返回-1
	int allocate(const int count);
	// 释放若干页的空间
	void release(const int address, const int amount);
};
```

`AddressPool` 的实现放在 `src/utils/address_pool.cpp` 中，如下所示。

```
AddressPool::AddressPool()
{
}

// 设置地址池BitMap
void AddressPool::initialize(char *bitmap, const int length, const int startAddress)
{
	resources.initialize(bitmap, length);
	this->startAddress = startAddress;
}

// 从地址池中分配count个连续页
int AddressPool::allocate(const int count)
{
	dword start = resources.allocate(count);
	return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
}

// 释放若干页的空间
void AddressPool::release(const int address, const int amount)
{
	resources.release((address - startAddress) / PAGE_SIZE, amount);
}
```

代码逻辑较为简单，这里便不再赘述。

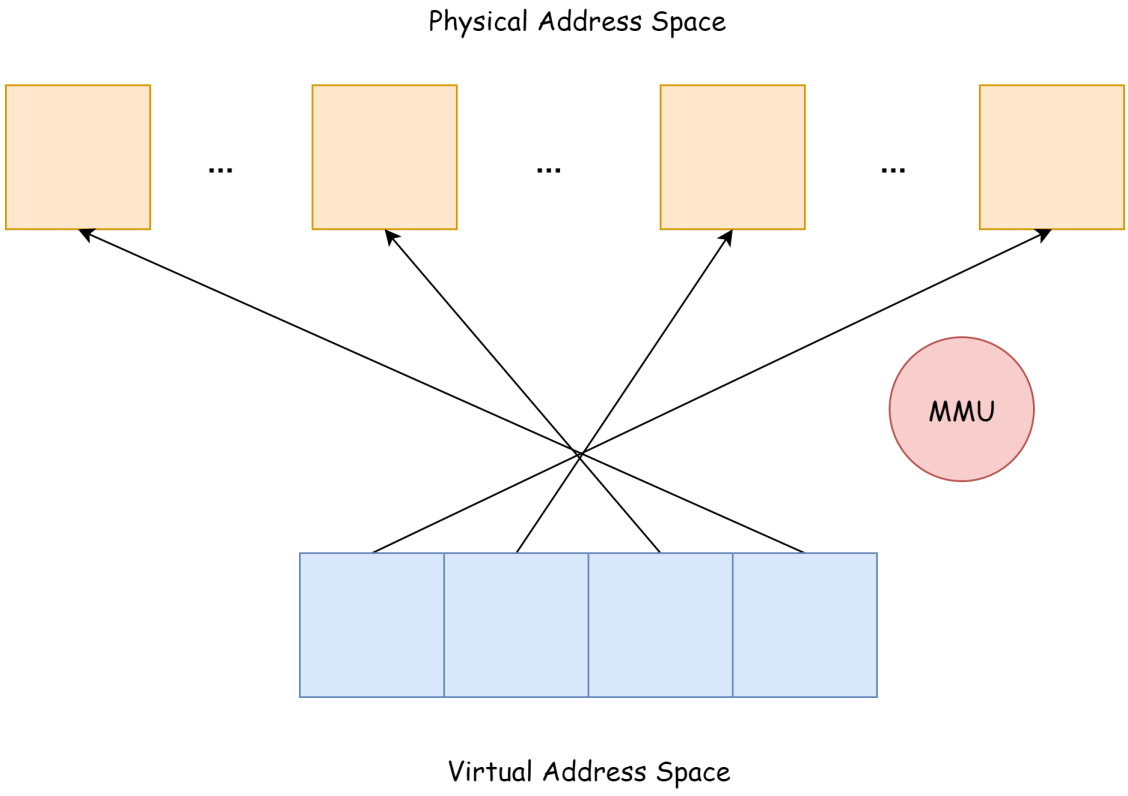
页内存分配

> 自此，线性地址统一使用虚拟地址来代替。

我们已经知道，开启了分页机制后，程序运行的地址是虚拟地址，虚拟地址都需要经过二级页表转换成物理地址后CPU才可以正常访问指令和数据。由此而产生的效果是，连续的虚拟地址对应不连续的物理地址，这种对应关系由二级页表来维护。

由于我们同时需要处理两个地址空间的内容，当我们进行页内存分配时，需要分别标识虚拟地址的分配状态和物理地址的分配状态，由此而产生了两个地址池——虚拟地址池和物理地址池。当我们需要进行连续的页内存分配时，我们先从虚拟地址池中取出连续的多个虚拟页，注意，虚拟页之间的虚拟地址是连续的。之后，我们从物理地址池中为每一个虚拟页分配相应大小的物理页，然后在二级页表中建立虚拟页和物理页之间的对应关系。此时，由于分页机制的存在，物理页的地址可以不连续。CPU的MMU会在程序执行过程中将虚拟地址翻译成物理地址。

例如，我们需要8个页的内存块。所以我们先从虚拟地址池中取出8个页的虚拟内存块的起始地址，此时8个页的虚拟地址是连续的。然后我们从物理地址池中先分配1页，建立和第1个虚拟页的对应关系；从物理地址池中分配1页，建立和第2个虚拟页的对应关系……如此进行下去，直到第8个虚拟页和物理页的对应关系被建立。显然，被分配的物理页的地址可以不连续，因此也就实现了连续的虚拟内存块对应不连续的物理内存块，如下所示。



接下来我们就来实现页内存分配。

Assignment 2 实现页内存分配

在实现页内存分配之前，我们首先要明白我们可以管理的内存的大小。内存的大小可以在实模式下通过BIOS中断来获取，但bochs默认的内存大小是32MB，为了简便起见，我直接指定了内存的大小为32MB，如下所示，定义放在 `include/os_constant.h` 中。

```
#define MEMORY_SIZE 0x2000000
#define BITMAP_START_ADDRESS 0x10000
#define KERNEL_VIRTUAL_START 0xc0100000
```

后面两个宏定义我们后面再讲解。

然后我们根据上面页内存分配的过程向 `MemoryManager` 加入如下成员函数。

```
enum AddressPoolType
{
	USER,
	KERNEL
};

class MemoryManager
{
private:
	// 内核物理地址池
	AddressPool kernelPhysical;
	// 用户物理地址池
	AddressPool userPhysical;
	// 内核虚拟地址池
	AddressPool kernelVirtual;

public:
	// 开启分页机制
	void openPageMechanism();
	// 初始化地址池
	void initialize(const int TOTAL_MEMORY);
	// 分配count个连续的页地址空间并返回起始地址
	void *allocatePages(enum AddressPoolType type, const dword count);
	// 释放空间
	void releasePages(const dword virtualAddress, const dword count);

private:
	// 从虚拟地址池中分配count个连续页
	static void *allocateVirtualPages(enum AddressPoolType type, const dword count);
	// 从物理地址池中分配1个页
	static void *allocatePhysicalPage(enum AddressPoolType type);
	// 建立虚拟地址和物理地址的联系
```

```
33          static bool connectPhysicalVritualPage(const dword virtualAddress, const dword
    physicalPageAddress);
34          // 获取virtualAddress对应的PDE虚拟地址
35          static dword *toPDE(const dword virtualAddress);
36          // 获取virtualAddress对应的PTE虚拟地址T
37          static dword *toPTE(const dword virtualAddress);
38          // 返回虚拟地址对应的物理地址
39          dword vaddr2paddr(dword vaddr);
40          // 为指定的虚拟地址分配物理地址
41          void *specifyPaddrForVaddr(enum AddressPoolType type, const dword vaddr);
42          // 释放物理地址
43          void releasePhysicalPage(dword address);
44          // 释放虚拟地址
45          void releaseVirtualPage(dword address);
46  };
```

虽然一下子加入的成员有点多😅，但万丈高楼平地起，我们接下来一步步地分析这些成员的实现和它们之间的联系。

我们后面会实现用户进程，用户进程实际上是运行中的程序。我们将程序编译好之后就放在文件系统中，当我们需要执行这个程序时，我们会为这个程序创建进程，然后分配内存，建立程序的线性地址到内存物理地址的映射，最后再将程序复制到内存中的，调度算法调度执行。因此用户地址的线性地址是从0开始的，但为什么不会与我们的内核发生冲突呢？这个因为用户进程有自己的页目录表和页表，即使用户进程和内核的线性地址相同，由于映射关系不同，也可以得到互不相关的物理地址。此时，我们把用户进程的线性地址空间称为用户虚拟地址空间，内核的线性地址空间称为内核虚拟地址空间。注意，内核虚拟地址空间只有一个，但不同的进程对应不同的用户虚拟地址空间。

因此，我们实际上有4个空间，用户虚拟地址空间，用户物理地址空间，内核虚拟地址空间，内核物理地址空间。但是，每一个进程都有自己的用户虚拟地址空间，因此并不是全局的，而是通过进程本身来管理。内存管理器`MemoryManager`只需要关心全局的地址空间即可，即用户物理地址空间，内核虚拟地址空间，内核物理地址空间，由此而产生了3个地址池`kernelPhysical`，`kernelVirtual`和`userPhysical`。

为了管理内存空间，我们首先需要初始化内存管理器，初始化内存管理器需要用到内存的大小`TOTAL_MEMORY`，如下所示，`initialize`并不关心`TOTAL_MEMORY`是通过中断获取的还是直接指定的。

```
1   void MemoryManager::initialize(const int TOTAL_MEMORY)
2   {
3           int usedMemory = 256 * PAGE_SIZE + 0x100000;
4           int freeMemory = TOTAL_MEMORY - usedMemory;
5
6           int freePages = freeMemory / PAGE_SIZE;
7           int kernelPages = freePages / 2;
8           int userPages = freePages - kernelPages;
9
10          int kernelPhysicalStartAddress = usedMemory;
```

```
11          int userPhysicalStartAddress = usedMemory + kernelPages * PAGE_SIZE;
12
13          int kernelPhysicalBitMapStart = BITMAP_START_ADDRESS;
14          int userPhysicalBitMapStart = kernelPhysicalBitMapStart + ceil(kernelPages, 8);
15          int kernelVirtualBitMapStart = userPhysicalBitMapStart + ceil(userPages, 8);
16
17          kernelPhysical.initialize((char *)kernelPhysicalBitMapStart, kernelPages,
    kernelPhysicalStartAddress);
18          userPhysical.initialize((char *)userPhysicalBitMapStart, userPages,
    userPhysicalStartAddress);
19          kernelVirtual.initialize((char *)kernelVirtualBitMapStart, kernelPages,
    KERNEL_VIRTUAL_START);
20
21          printf("kernel pool\n     start address: %x\n     total pages: %d\n     bit map start
    address: %x\n",
22                      kernelPhysicalStartAddress, kernelPages, kernelPhysicalBitMapStart);
23
24          printf("user pool\n     start address: %x\n     total pages: %d\n     bit map start
    address: %x\n",
25                      userPhysicalStartAddress, userPages, userPhysicalBitMapStart);
26
27          printf("kernel virtual pool\n     start address: %x\n     total pages: %d\n     bit map
    start address: %x\n",
28                      KERNEL_VIRTUAL_START, userPages, kernelVirtualBitMapStart);
29      }
30      }
```

第3行，我们假设内核很小，只放在0~1MB的内存物理地址空间中，并且将0~1MB的内存固定分配给内核，因此我们已经使用了1MB的内存。但是，`256 * PAGE_SIZE`这256个页表是从哪里来的呢？注意到我们的页目录表放在了`0x100000`处，起始地址为`0x101000`的页表存放了指向物理地址0~1MB的物理页的256个页表项。但是，另外254个页表是用来做什么的呢？

我们已经知道，用户进程有自己的独立虚拟地址空间。但是，用户进程如果想要通信，那么这些用户进程之间就需要有一块共享的公共区域，这个共享的公共区域的区域是内核虚拟地址空间。为此，我们将内核虚拟地址空间映射到用户虚拟地址空间的3GB~4GB范围内。这样，每一个用户进程的3GB~4GB范围都是共享的内核空间。注意到，3GB~4GB的虚拟地址空间对应的页目录项是第768~1023个页目录项，这就是为什么我们在建立分页机制的时使第768个页目录项和第0个页目录项相同，目的是将内核复制到3GB~4GB的范围。我们只需要保持虚拟地址3GB~4GB对应的页目录项和内核真正的虚拟地址对应的页目录项相同即可，此时，对于任意一个真正的内核虚拟地址`address`，虚拟地址`address+0xc0000000`和`address`具有相同的物理地址。

注意到768~1023之间总共有256个页目录项，而第1023个页目录项指向了页目录表本身，因此只有255个页表项。为了方便，我将这255个页目录项指向的页表安排在了页目录表之后，于是，255个页表加上1个页目录表恰好是256个页表。

此时，我们可分配的内存 `freeMemory` 等于总内存 `TOTAL_MEMORY` 减去已经被固定分配的内存 `usedMemory` 。接着，我们将剩下的内存空间划分成两半，前半部分给内核物理地址空间，后半部分给用户地址物理空间。因此，内核物理地址空间地起始地址就是 `usedMemory` ，用户物理地址空间紧跟内核物理地址空间，起始地址便为 `usedMemory + kernelPages * PAGE_SIZE` 。

我们将三个地址池所用的位图也放到0~1MB的内核空间中，起始地址是 `BITMAP_START_ADDRESS=0x10000` ，然后依次放置内核物理地址空间的位图，用户物理地址空间的位图和内核虚拟地址空间的位图。

最后，我们使用前面得到的结果来初始化三个地址池即可。由于0~1MB的内核空间已经占据了虚拟地址 `0x100000` 以下的空间，因此我们定义内核虚拟地址空间的起始地址是 `0x100000` 。但我们希望所有进程都能够共享内核分配的页内存，因此我们把起始地址加上 `0xc0000000` 提升到3GB~4GB的虚拟地址空间中，也就是 `0xc0100000` 。

此时，我们完成了内存管理器的初始化。

初始化内存管理器之后，我们就可以实现页内存分配了。由于我们使用了分页机制，我们的程序使用的是虚拟地址空间的地址，但虚拟地址总是要经过分页机制变换到物理地址。也就是说，对每一个虚拟地址，我们都要为其指定一个对应的物理地址。

因此，页内存分配分为以下3步。

- 从虚拟地址池中分配若干连续的虚拟页。
- 对每一个虚拟页，从物理地址池中分配1页。
- 为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。

因此，一个页内存分配的函数如下所示。

```
int allocatePages(enum AddressPoolType type, const int count)
{
      int virtualAddress = allocateVirtualPages(type, count);
      if (!virtualAddress)
      {
            return 0;
      }

      bool flag;
      int physicalPageAddress;
      int vaddress = virtualAddress;

      for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
      {
            flag = true;
            physicalPageAddress = allocatePhysicalPages(type, 1);
            if (physicalPageAddress)
            {
                  flag = connectPhysicalVritualPage(vaddress, physicalPageAddress);
            }
```

```
            else
            {
                flag = false;
            }

            if (flag)
            {
                releasePages(type, virtualAddress, i);
                releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
                return 0;
            }
        }

        return virtualAddress;
}
```

注意，当我们在为连续的虚拟页逐个分配物理页时，若遇到无法分配物理页的情况，那么说明我们无法分配所需的内存空间，此时我们应该返回0。但是，由于我们是逐步地为每一个虚拟页建立到一个物理页的映射关系，因此在返回前，之前成功分配的虚拟页和物理页都要释放。否则就会造成内存泄漏，这部分内存无法再被分配。

此时，我们分配的虚拟地址是连续的，但虚拟页对应的物理页可以是不连续的。通过分页机制，我们可以把连续的地址变换到不连续的地址中，这就是分页机制的精妙之处。

下面我们分别来看页内存分配的每个步骤。

从虚拟地址池中分配若干连续的虚拟页。虚拟页的分配通过函数 `allocateVirtualPages` 来实现，如下所示。

```
int allocateVirtualPages(enum AddressPoolType type, const int count)
{
        int start = -1;

        if (type == AddressPoolType::KERNEL)
        {
                start = kernelVrirtual.allocate(count);
        }

        return (start == -1) ? 0 : start;
}
```

由于我们没有实现用户进程，此时能够分配页内存的地址池只有内核虚拟地址池。因此，对于其他类型的地址池，一律返回0，即分配失败。

对每一个虚拟页，从物理地址池中分配1页。物理页的分配通过函数 `allocatePhysicalPages` 来实现，如下所示。

```
int allocatePhysicalPage(enum AddressPoolType type, const int count)
{
	int start = -1;

	if (type == AddressPoolType::KERNEL)
	{
		start = kernelPhysical.allocate(count);
	}
	else if (type == AddressPoolType::USER)
	{
		start = userVirtual.allocate(count);
	}

	return (start == -1) ? 0 : start;
}
```

我们的物理地址池有两个，用户物理地址池和内核物理地址池，因此在分配物理页的时候也应该区别对待。

为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。建立虚拟页到物理页的映射关系通过函数 `connectPhysicalVritualPage` 来实现，如下所示。

```
bool connectPhysicalVritualPage(const int virtualAddress, const int physicalPageAddress)
{
	// 计算虚拟地址对应的页目录项和页表项
	int *pde = toPDE(virtualAddress);
	int *pte = toPTE(virtualAddress);

	if (*pde & 0x00000001) // 页目录项有对应的页表
	{
		// 使页表项指向物理页
		*pte = physicalPageAddress | 0x7;
	}
	else // 页目录项无对应的页表
	{
		// 从内核物理地址空间中分配一个页表
		int page = allocatePhysicalPage(AddressPoolType::KERNEL);
		if (!page)
			return false;

		// 使页目录项指向页表
		*pde = page | 0x7;
		// 初始化页表
		char *pagePtr = (char *)(((int)pte) & 0xfffff000);
		memset(pagePtr, 0, PAGE_SIZE);
		// 使页表项指向物理页
		*pte = physicalPageAddress | 0x7;
```

```
26        }
27
28        return true;
29    }
```

在建立虚拟页到物理页的映射关系之前，我们首先来回忆一下在二级分页机制下，虚拟地址是如何变换到物理地址的。考虑一个虚拟地址$virtual$，变换过程如下所示。

- CPU先取虚拟地址的31-22位$virtual[31:22]$，在cr3寄存器中找到页目录表的物理地址，然后根据页目录表的物理地址在页目录表中找到序号为$virtual[31:22]$的页目录项，读取页目录项中的页表的物理地址。
- CPU再取虚拟地址的21-12位$virtual[21:12]$，根据第一步取出的页表的物理地址，在页表中找到序号为$virtual[21:12]$的页表项，读取页表项中的物理页的物理地址$physical$。
- 用物理页的物理地址的的31-12位$physical[31:12]$替换虚拟地址的31-12位$virtual[31:12]$得到的结果就是虚拟地址对应的物理地址。

注意，只有程序才会使用虚拟地址，cr3寄存器，页目录项，页表项和CPU寻址中的地址都是物理地址。

实际上，页内存分配就是为若干个连续的虚拟页指定物理页的过程。而CPU是根据页目录表和页表的内容来找到虚拟页对应的物理页的。因此，为了给一个虚拟页指定物理页，我们只需要修改虚拟页地址对应的页表项和页目录项即可。注意，一个虚拟页中的虚拟地址对应的相同的页目录项和页表项。

为了修改页目录项和页表项的物理地址，我们首先要构造出虚拟地址对应的页目录项和页表项虚拟地址。这里，同学们会有疑问——“我们已经知道了页目录表的物理地址，为什么不直接使用这个物理地址呢？”。这是因为程序使用的是虚拟地址，即便我们使用的是物理地址，这个“物理地址”还是会经过找页目录项，找页表项，物理地址替换三个步骤来变换成真正的物理地址，但这样做就出错了。这就回到了我之前强调的问题上——同学们在开启分页机制后，对于一个地址，一定要想清楚是虚拟地址还是物理地址，如果是虚拟地址，则需要拆分成三部分来看待，绝对不可以当成物理地址来看待。这一点在刚开始接触的时候理解起来比较抽象，这也是我开篇所说的“所见非所得”的含义。

> what you see is not what you get.

因此，为了访问虚拟地址对应的页目录项和页表项，我们需要根据页目录项和页表项将页目录项和页表项的虚拟地址构造出来。对于一个虚拟地址，我们需要构造三部分的信息，页目录号$virtual[31:22]$，页号$virutal[21:12]$和偏移地址$virtual[11:0]$。

考虑虚拟地址$virtual$，我们下面首先来看页目录项pde(Page Directory Entry)的构造。

注意到我们之前在开启分页机制的时候，我们设置最后一个页目录项指向了页目录表，这样做就是为了构造页目录项和页表项的虚拟地址而服务的。首先，页目录项所在的物理页是页目录表，$virtual$的页目录号是$virtual[31:22]$，而每一个页目录项的大小是4个字节，因此我们有

$$pde[11:0] = 4 \times virtual[31:22]$$

接下来我们构造$pde[21:12]$。pde是位于页目录表的，那么，在现有的页表中，哪一个页表的哪一个页表项指向了页目录表呢？实际上，页目录表是页表的一种特殊形式，页目录表的第1023个页目录项指向了页目录表。因此我们有

$$pde[21:12] = 1111111111_2$$

其中，下标2表示二进制，$2^{10} - 1 = 1023$。

最后，我们来构造$pde[31:22]$。我们实际上把页目录表当成了页表，而$pde[31:22]$是页目录项的序号，我们需要知道页目录表的哪一个页表项指向了这个“页表”显然，答案是第1023个页表项，因此我们有

$$pde[31:22] = 1111111111_2$$

至此，我们已经完成了页目录项的构建。上面构造的过程是从物理地址到虚拟地址的，我们现在来看一个从虚拟地址到物理地址变换的例子来加深对上述过程的理解。假设我们需要构造虚拟地址`0x2a49fe12`对应的页目录项，分下面三步完成。

$$\begin{aligned}
&virtual[31:22] = 0xa9 = 169\\
&virtual[21:12] = 0x9f = 159\\
&pde[11:0] = 4 \times virutal[31:22] = 4 \times 159 = 0x27c\\
&pde[21:12] = 0x3ff\\
&pde[31:22] = 0x3ff
\end{aligned}$$

因此，pde的值为$0xfffff27c$，然后我们将这个虚拟地址转换到物理地址。

- 先取虚拟地址的31-22位$pde[31:22]$，在cr3寄存器中找到页目录表的物理地址，然后根据页目录表的物理地址在页目录表中找到序号为$pde[31:22]$的页目录项，读取页目录项中的页表的物理地址。取出的页目录项是第1023个页目录项，这个页目录项指向了页目录表。
- CPU再取虚拟地址的21-12位$pde[21:12]$，根据第一步取出的页表的物理地址，在页表中找到序号为$pde[21:12]$的页表项，读取页表项中的物理页的物理地址$physical$。此时的页表是页目录表，读取的页表项是第1023个页目录项，这个页表项指向了页目录表。
- 用物理页的物理地址的的31-12位$physical[31:12]$替换虚拟地址的31-12位$pde[31:12]$得到的结果就是虚拟地址对应的物理地址。此时，$physical$就是页目录表的物理地址。

然后，我们再来构建页表项pte。

首先，页表项所在的物理页是页表，$virtual$的页号是$virtual[21:12]$，而每一个页表项的大小是4个字节，因此我们有

$$pte[11:0] = 4 \times virtual[21:12]$$

接下来我们构造$pte[21:12]$。pte是位于页表的，那么，在现有的页表中，哪一个页表的哪一个页表项指向了pte所在的页表呢？回忆起二级分页机制的地址变换过程我们发现，页目录表的第$virutal[31:22]$页目录项指向了这个页表，因此我们有

$$pte[21:12] = virtual[31:22]$$

最后，我们来构造$pde[31:22]$。我们实际上把页目录表当成了页表，而$pte[31:22]$是页目录项的序号，我们需要知道页目录表的哪一个页表项指向了这个“页表”显然，答案是第1023个页表项，因此我们有

$$pte[31:22] = 1111111111_2$$

写成代码如下所示。

```
int toPDE(const int virtualAddress)
{
	return (0xfffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}

int toPTE(const int virtualAddress)
{
	return (0xffc00000 + ((virtualAddress & 0xffc00000) >> 10) + (((virtualAddress & 0x003ff000) >> 12) * 4));
}
```

找到虚拟地址对应的pde和pte后，我们便可以建立虚拟页到物理页的映射。首先看pde中是否有对应的页表，如果没有，就要先分配一个物理页，然后初始化这新分配的物理页并将其地址写入pde，以作为pde指向的页表。在pde对应的物理页存在的前提下，将之前为虚拟页分配的物理页地址写入pte即可。

但我们在分配页内存时，如果遇到物理页无法分配的情况，之前成功分配的虚拟页和物理页都要释放。否则就会造成内存泄漏，这部分内存无法再被分配，释放页内存如下。

```
void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)
{
	int vaddr = virtualAddress;
	int *pte, *pde;
	bool flag;
	const int ENTRY_NUM = PAGE_SIZE / sizeof(int);

	for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
	{
		releasePhysicalPages(type, vaddr2paddr(vaddr), 1);

		// 设置页表项为不存在，防止释放后被再次使用
		pte = (int *)toPTE(vaddr);
		*pte = 0;
	}
```

```
16
17          releaseVirtualPages(type, virtualAddress, count);
18  }
```

实际上页内存的释放是逆页内存分配的过程，分2个步骤完成。

- 对每一个虚拟页，释放为其分配的物理页。
- 释放虚拟页。

我们分别来看上面两个步骤。

对每一个虚拟页，释放为其分配的物理页。首先，由于物理地址池存放的是物理地址，为了释放物理页，我们要找到虚拟页对应的物理页的物理地址，如下所示。

```
1   int MemoryManager::vaddr2paddr(int vaddr)
2   {
3           int *pte = (int *)toPTE(vaddr);
4           int page = (*pte) & 0xfffff000;
5           int offset = vaddr & 0xfff;
6           return (page + offset);
7   }
```

根据分页机制，一个虚拟地址对应的物理页的地址是存放在页表项中的。因此，我们先求出虚拟地址的页表项的虚拟地址，然后访问页表项，取页表项内容的31-12位就是物理页的物理地址，最后替换虚拟地址的31-12位即可得到虚拟地址对应的物理地址。

然后我们将这个物理地址归还到物理地址池，如下所示。

```
1   void MemoryManager::releasePhysicalPages(enum AddressPoolType type, const int paddr, const int
    count)
2   {
3           if (type == AddressPoolType::KERNEL)
4           {
5                   kernelPhysical.release(paddr, count);
6           }
7           else if (type == AddressPoolType::USER)
8           {
9
10                  userPhysical.release(paddr, count);
11          }
12  }
```

归还了物理地址后，我们就要将虚拟页对应的页表项置0，这是为了防止在虚拟页释放后被再次寻址。

释放虚拟页。释放虚拟页的函数如下所示。

```
void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count)
{
	if (type == AddressPoolType::KERNEL)
	{
		kernelVirtual.release(vaddr, count);
	}
}
```

我们现在还没实现用户进程，而每一个用户进程都有自己独立的虚拟地址池，因此这里我们处理内核虚拟地址池中的地址，等到我们实现了用户进程后再修改这部分的代码。

至此，页内存分配的实现已经完成，同学们可以自行测试。

内核线程

程序、进程和线程

程序、进程和线程的区别常常令人混淆，这里我给出我对这三者的理解。

- 程序是指静态的、存储在文件系统上的、尚未运行的指令代码，它是实际运行时的程序的映像。
- 进程是指正在运行的程序，即进行中的程序，程序必须在获得运行所需要的各类资源后才可以成为进程，资源包括进程所使用的栈，寄存器等。
- 线程实际上是函数的载体，属于创建它的进程。进程所创建的所有线程共享进程所拥有的全部资源。

进程和线程的关系

线程出现的目的是为了加速进程的执行。例如，一个程序想统计A字符串中字母，统计B字符串中数字。显然，这两个过程是独立的，也就是说我们可以同时对A、B字符串进行统计。此时，我们可以把这两个过程写成函数，然后放到线程中并发执行。但是，在线程出现之前，我们不得不先统计A字符串中字母，然后再统计B字符串中字母。相比之下，进程中的线程的并发执行的确加速了进程的运行速度。

在线程出现之前，处理器调度的单元是进程。进程中包含整个程序的执行代码，运行资源等。处理器依据某种方法改变各个进程的状态如阻塞态，就绪态，运行态等来管理系统中的多个进程。也就是说，处理器实际上管理的是进程对应的整个程序。在线程出现后，如果处理器的调度单元依然是进程，这样非但没有实现线程的并发执行，而且当处理器阻塞某个进程时，进程所创建的所有线程都会被阻塞。所以，处理器的调度单元变为了线程。但是，现在问题来了，一个进程可能不会创建线程，那么处理器是否无法对这个进程进行管理了呢？其实不是的，我们已经知道，一个C/C++程序总是会从main函数开始执行，而线程恰好是函数的载体。于是，只要创建线程运行这个main函数，我们就相当于执行了这个进程。因此，在线程出现后，为了实现对进程的管理，一个进程中至少会包含一个线程，此时这个进程被称为“单线程进程”，而主动创建线程的进程被称为“多线程进程”。为了区别线程出现前的

进程，我将线程出现前的进程称为“传统型线程”，将线程出现后的进程直接称为“进程”。因此，一个进程至少是一个“单线程进程”，进程是资源的拥有者，线程是资源的使用者，用公式表示如下。

$$进程 = 线程 + 资源.$$

用户线程和内核线程

用户线程指线程只由用户进程来实现，操作系统中无线程机制。在用户空间中实现线程的好处是可移植性强，由于是用户级的实现，所以在不支持线程的操作系统上也可以写出完美支持线程的用户程序。但是，用户线程存在以下缺点。

- 进程中的某个线程若出现了阻塞（通常是由于系统调用造成的）， 操作系统不知道进程中存在线程，它以为此进程是传统型进程（单线程进程），因此会将整个进程挂起，即进程中的全部线程都无法运行。
- 线程未在内核空间中实现，因此对于操作系统来说，调度器的调度单元是整个进程，并不是进程中的线程，所以时钟中断只能影响进程一级的执行流。当时钟中断发生后，操作系统的调度器只能感知到进程一级的调度实体，它要么把处理器交给进程 A，要么交给进程 B，绝不可能交给进程中的某个线程 。因此，但凡进程中的某个线程开始在处理器上执行后，只要该线程不主动让出处理器，此进程中的其他线程都没机会运行。同时， 由于整个进程占据处理器的时间片是有限的， 这有限的时间片还要再分给内部的线程，所以每个线程执行的时间片非常非常短暂，再加上进程内线程调度器维护线程表、运行调度算法的时间片消耗，反而抵销了内部调度带来的提速。

所以，为了最大地发挥线程的价值，我们需要在内核中实现线程机制。内核线程具有以下特点。

- 相比在用户空间中实现线程，内核提供的线程相当于让进程多占了处理器资源，比如系统中运行有进程 A和一传统型进程B，此时进程A中显式创建了3个线程，这样一来，进程A加上主线程便有了4个线程，加上进程B，内核调度器眼中便有了5个独立的执行流，尽管其中4个都属于进程A，但对调度器来说这4个线程和进程一样被调度，因此调度器调度完一圈后，进程A使用了 80%的处理器资源，这才是真正的提速。
- 另一方面的优点是当进程中的某一线程阻塞后，由于线程是由内核空间实现的，操作系统认识线程，所以就只会阻塞这一个线程，此线程所在进程内的其他线程将不受影响，这又相当于提速了。
- 用户进程需要通过系统调用陷入内核，这多少增加了一些现场保护的栈操作，这还是会消耗一些处理器时间，但和上面的大幅度提速相比，这显然是微不足道的。

线程的描述

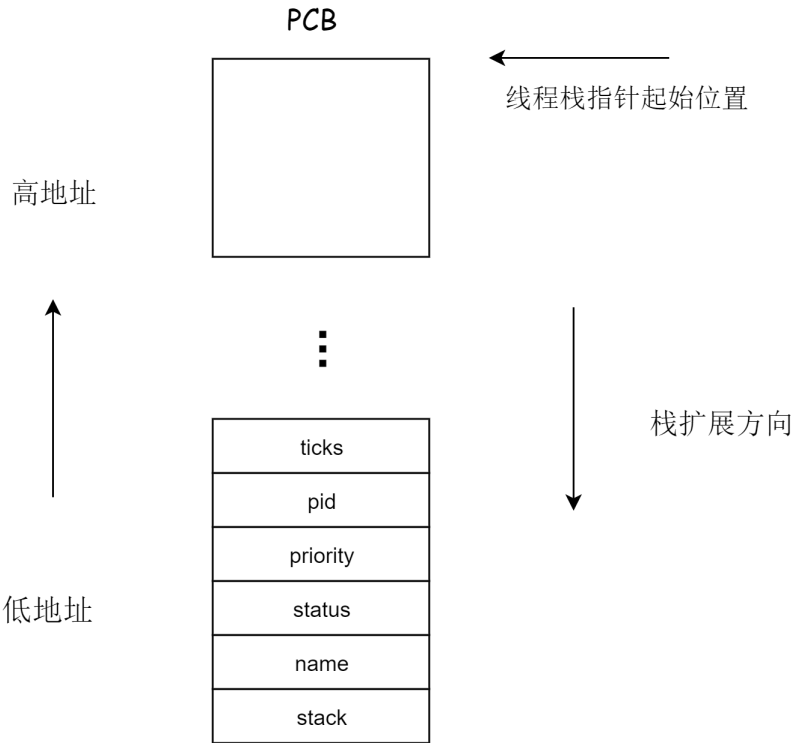
线程的组成部分有操作系统为线程分配的栈，状态，优先级，运行时间，线程负责运行的函数，函数的参数等，这些组成部分被集中保存在一个结构中——PCB(Process Control Block)，如下所示，代码放在 `include/thread.h` 中。

```
1  enum ThreadStatus
2  {
3          CREATE,
```

```
4         RUNNING,
5         READY,
6         BLOCKED,
7         DEAD
8   };
9
10  struct PCB
11  {
12        int *stack;                              // 栈指针，用于调度时保存esp
13        char name[MAX_PROGRAM_NAME + 1]; // 线程名
14        enum ThreadStatus status;           // 线程的状态
15        int priority;                            // 线程优先级
16        int pid;                                  // 线程pid
17        int ticks;                               // 线程时间片总时间
18        int ticksPassedBy;                  // 线程已执行时间
19        ListItem tagInGeneralList; // 线程队列标识
20        ListItem tagInAllList;       // 线程队列标识
21  };
```

我们来看PCB各成员的含义。

- `stack`。各个内核线程是共享内核空间的，但又相对独立，这种独立性体现在每一个线程都有自己的栈。那么线程的栈保存在哪里呢？线程的栈就保存在线程的TCB中，我们会为每一个TCB分配一个页，上面的`struct TCB`只是这个页的低地址部份，线程的栈指针从这个页的结束位置向下递减，如下所示。



因此，我们不能向线程的栈中放入太多的东西，否则当栈指针向下扩展时，会与线程的TCB的信息发生覆盖，最终导致错误。`stack`的作用是在线程被换下处理器时保存esp的内容，然后当线程被换上处理器后，我们用`stack`去替换esp的内容，从而实现恢复线程运行的效果。

- `status`是线程的状态。

- `name` 是线程的名称。
- `priority` 是线程的优先级，线程的优先级决定了抢占式调度的过程和线程的执行时间。
- `pid` 是线程的标识符，每一个线程的pid都是唯一的。
- `ticks` 是线程剩余的执行次数，每发生中断一次记为一个 `tick`，当 `ticks=0` 时，线程会被换下处理器，然后将其他线程换上处理器执行。
- `ticksPassedBy` 是线程总共执行的次数。
- `tagInGeneralList` 和 `tagInAllList` 是线程在线程队列中的标识，用于在线程队列中找到线程的PCB。

其中，`ListItem` 是链表 `List` 中的元素的表示，如下所示，代码放置在 `list.h` 中。

```
struct ListItem
{
    ListItem *previous;
    ListItem *next;
};
```

`List` 是一个带头节点的双向链表，定义如下。

```
class List
{
public:
    ListItem head;

public:
    // 初始化List
    List();
    // 显式初始化List
    void initialize();
    // 返回List元素个数
    int size();
    // 返回List是否为空
    bool empty();
    // 返回指向List最后一个元素的指针
    // 若没有，则返回nullptr
    ListItem *back();
    // 将一个元素加入到List的结尾
    void push_back(ListItem *itemPtr);
    // 删除List最后一个元素
    void pop_back();
    // 返回指向List第一个元素的指针
    // 若没有，则返回nullptr
    ListItem *front();
    // 将一个元素加入到List的头部
```

```
26          void push_front(ListItem *itemPtr);
27          // 删除List第一个元素
28          void pop_front();
29          // 将一个元素插入到pos的位置处
30          void insert(int pos, ListItem *itemPtr);
31          // 删除pos位置处的元素
32          void erase(int pos);
33          void erase(ListItem *itemPtr);
34          // 返回指向pos位置处的元素的指针
35          ListItem *at(int pos);
36          // 返回给定元素在List中的序号
37          int find(ListItem *itemPtr);
38  };
```

关于 `List` 的成员函数的实现同学们应该倒背如流，这里我便不班门弄斧了，实现代码放置在 `src/utils/list.cpp` 中。

TCB相当于线程的身份证，只要掌握了TCB就能掌握线程的全部信息，下面我们来创建线程。

线程的创建

首先，我们创建一个程序调度器 `ProgramManager`，用于线程和进程的调度，如下所示，代码放置在 `include/program.h` 中。

```
1   #ifndef PROGRAM_H
2   #define PROGRAM_H
3
4   #include "list.h"
5   #include "thread.h"
6
7   class ProgramManager
8   {
9   public:
10          List allPrograms;    // 所有状态的线程/进程的队列
11          List readyPrograms; // 处于ready(就绪态)的线程/进程的队列
12          PCB *running;          // 当前执行的线程
13  public:
14          ProgramManager();
15          void initialize();
16          // 创建一个线程并放入就绪队列
17          // 成功，返回pid；失败，返回-1
18          int executeThread(ThreadFunction function, void *parameter, const char *name, int
    priority);
19          // 执行一次调度
20          void schedule();
21
22  public:
```

```
        //  分配一个未被占用的pid
        int allocatePid();
        // 按pid查找线程/进程
        PCB *findProgramByPid(int pid);
        // ListItem转换成PCB
        PCB *ListItem2PCB(ListItem *item);
};

void program_exit();

#endif
```

`allPrograms` 是所有状态的线程和进程的队列，其中放置的是这些线程和进程的 `tagInAllList`。 `readyPrograms` 是处在ready(就绪态)的线程/进程的队列，放置的是 `tagInGeneralList`。

然后我们在 `include/os_modules` 中定义一个全局的 `ProgramManager`，如下所示。

```
#ifndef OS_MODULES_H
#define OS_MODULES_H

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

// 中断管理器
InterruptManager interruptManager;
// 输出管理器
STDIO stdio;
// 内存管理器
MemoryManager memoryManager;
// 进程/线程管理器
ProgramManager programManager;

#endif
```

在使用 `ProgramManager` 的成员函数前，我们必须初始化 `ProgramManager`，如下所示，代码放置在 `src/program/program.cpp` 中。

```
ProgramManager::ProgramManager()
{
	initialize();
}

void ProgramManager::initialize()
{
	allPrograms.initialize();
	readyPrograms.initialize();
	runningThread = nullptr;
}
```

现在我们来创建线程。

前面提到，线程是一个函数的载体，因此线程执行的代码实际上是一个函数的代码。那么线程可以执行哪些函数的代码呢？这里我们规定线程只能执行返回值为 `void`，参数为 `void *` 的函数，其中，`void *` 是函数参数的指针。因此，当我们有多个参数时，我们需要将其放入到一个结构体中，线程执行的函数类型声明如下。

```
typedef void(*ThreadFunction)(void *);
```

线程的创建如下所示。

```
int ProgramManager::executeThread(ThreadFunction function, void *parameter, const char *name, int priority)
{
	bool status = interruptManager.getInterruptStatus();
	interruptManager.disableInterrupt();

	PCB *thread = (PCB *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
	if (!thread)
		return -1;

	memset((char *)thread, 0, PAGE_SIZE);

	for (int i = 0; i < MAX_PROGRAM_NAME && name[i]; ++i)
	{
		thread->name[i] = name[i];
	}

	thread->status = ThreadStatus::READY;
	thread->priority = priority;
	thread->ticks = priority * 10;
	thread->ticksPassedBy = 0;

```

```
    thread->pid = allocatePid();
    if (thread->pid == -1)
    {
        memoryManager.releasePages(AddressPoolType::KERNEL, (int)thread, 1);
        return -1;
    }

    // 线程栈
    thread->stack = (int *)((int)thread + PAGE_SIZE);
    thread->stack -= 7;
    thread->stack[0] = 0;
    thread->stack[1] = 0;
    thread->stack[2] = 0;
    thread->stack[3] = 0;
    thread->stack[4] = (int)function;
    thread->stack[5] = (int)program_exit;
    thread->stack[6] = (int)parameter;

    allPrograms.push_back(&(thread->tagInAllList));
    readyPrograms.push_back(&(thread->tagInGeneralList));

    interruptManager.setInterruptStatus(status);

    return thread->pid;
}
```

由于我们现在来到的多线程的环境，诸如内存分配的工作实际上都需要进行线程互斥处理，但我们并没有实现线程互斥的工具如锁、信号量等，因此这里我们只是简单地使用关中断和开中断来实现线程互斥。为什么开/关中断有效呢？在后面可以看到，我们是在时钟中断发生时来进行线程调度的，因此关中断后，时钟中断无法被响应，线程就无法被调度直到再次开中断。只要线程无法被调度，那么线程的工作也就无法被其他线程打断，因此就实现了线程互斥。

和开/关中断等相关的的函数定义在 `include/interrupt.h` 中，如下所示。

```
class InterruptManager
{

    ...

    // 开中断
    void enableInterrupt();
    // 关中断
    void disableInterrupt();
    // 获取中断状态
    // 返回true，中断开启；返回false，中断关闭
    bool getInterruptStatus();
    // 设置中断状态
    // status=true，开中断；status=false，关中断
```

```
15        void setInterruptStatus(bool status);
16
17        ...
18    };
```

函数的实现比较简单，放置在`src/interrupt/interrupt.cpp`中，这里便不再赘述。

关中断后，我们向内核空间申请一页作为线程的PCB，若申请失败，则返回`-1`表示线程创建失败。否则，我们先将PCB清0，然后设置PCB的成员`name`，`status`，`priority`，`ticks`和`ticksPassedBy`。然后我们为线程申请一个pid，申请pid的函数如下。

```
1   int ProgramManager::allocatePid()
2   {
3         bool status = interruptManager.getInterruptStatus();
4         interruptManager.disableInterrupt();
5
6         // 0号线程
7         if (allPrograms.empty())
8               return 0;
9
10        int pid = -1;
11        PCB *program;
12
13        for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
14        {
15              program = findProgramByPid(i);
16              if (!program)
17              {
18                    pid = i;
19                    break;
20              }
21        }
22
23        interruptManager.setInterruptStatus(status);
24
25        return pid;
26  }
```

`allocatePid`的基本实现思想是令pid从0到`MAX_PROGRAM_AMOUNT`（最大线程数量）开始遍历，检查pid是否和已有的线程的pid相同，若相同，则检查下一个pid，否则返回这个pid。检查pid是否相同是通过按pid查找线程的PCB的函数来完成的，这个函数如下所示。

```
1   PCB *ProgramManager::findProgramByPid(int pid)
2   {
3         bool status = interruptManager.getInterruptStatus();
```

```
4         interruptManager.disableInterrupt();
5
6         ListItem *item = allPrograms.head.next;
7         PCB *program, *ans;
8
9         ans = nullptr;
10        while (item)
11        {
12
13              program = (PCB *)(((dword)item) & 0xfffff000);
14
15              if (program->pid == pid)
16              {
17                    ans = program;
18                    break;
19              }
20              item = item->next;
21        }
22
23        interruptManager.setInterruptStatus(status);
24
25        return ans;
26  }
```

`findProgramByPid` 遍历包含了所有线程的线程队列 `allPrograms`，然后比较每一个线程的pid是否和给定的pid相同，若相同，则返回线程的PCB指针，否则返回nullptr。线程队列存储的不是各个线程的PCB，而是通过PCB的两个 `ListItem` 成员将它们串接成一个 `List`。为什么通过PCB的 `ListItem` 在线程队列中保存了PCB的信息呢？我们知道，`ListItem` 有两个 `ListItem *` 成员，`previous` 和 `next`。这两个成员分别保存了线程队列中上一个线程的 `ListItem` 的地址和下一个线程的 `ListItem` 的地址。由于PCB是一个页，起始地址的低12位必然为0(页的大小是4KB)，而PCB的 `ListItem` 成员是位于这个页当中的，因此，`ListItem` 成员的地址和 `0xfffff000` 相与便可得到PCB的地址。知道了PCB的地址就可以访问PCB的其他成员，这也就知道了PCB的所有信息。

> 通过ListItem将PCB存储在线程队列中的做法比较巧妙，同学们注意理解和体会。

如果我们找不到一个可用的pid，我们就要返回-1表明线程创建失败。但在返回前，我们需要释放PCB占用的页内存。如果找到可用的pid，我们就初始化线程的栈。线程的栈是从PCB的顶部开始向下增长的，因此，线程栈的初始地址是PCB的起始地址加上页的大小。然后，我们在栈中放入一些数值。

- 4个为0的值是要放到ebp，ebx，edi，esi中的。
- `thread->stack[4]` 是线程执行的函数的起始地址。
- `thread->stack[5]` 是线程的返回地址。
- `thread->stack[6]` 是线程的参数。

至于这4部份的作用我们在线程的调度中统一讲解。

创建完线程的PCB后，我们将其放入到 `allPrograms` 和 `readyPrograms` 中，等待时钟中断来的时候可以被调度上处理器。

最后我们将中断的状态恢复，此时我们便创建了一个线程。

线程的调度

我们首先要修改之前的处理时钟中断函数，如下所示。

```
void c_time_interrupt_handler()
{
	PCB *cur = programManager.running;
	//printf("pid %d ticks: %d\n", cur->pid, cur->ticks);
	if (cur->ticks)
	{
		--cur->ticks;
		++cur->ticksPassedBy;
	}
	else
	{
		programManager.schedule();
	}

}
```

这里，我们使用的是时间片轮转的线程调度算法。当时钟中断到来时，我们对当前线程的 `ticks` 减1，直到 `ticks` 等于0，然后执行线程调度。线程调度的代码如下。

```
void ProgramManager::schedule()
{
	bool status = interruptManager.getInterruptStatus();
	interruptManager.disableInterrupt();

	if (readyPrograms.size() == 0)
	{
		interruptManager.setInterruptStatus(status);
		return;
	}

	if (running->status == ThreadStatus::RUNNING)
	{
		running->status = ThreadStatus::READY;
		running->ticks = running->priority * 10;
		readyPrograms.push_back(&(running->tagInGeneralList));
	}
	else if (running->status == ThreadStatus::DEAD)
```

```
19          {
20                  memoryManager.releasePages(AddressPoolType::KERNEL, (int)running, 1);
21          }
22
23          PCB *cur = running->status == DEAD ? 0 : running;
24          ListItem *item = readyPrograms.front();
25          PCB *next = ListItem2PCB(item);
26          next->status = ThreadStatus::RUNNING;
27          running = next;
28          readyPrograms.pop_front();
29
30          //printf("schedule: %x, %x\n", cur, next);
31
32          asm_switch_thread(cur->stack, next->stack);
33
34          interruptManager.setInterruptStatus(status);
35  }
```

首先，为了实现线程互斥，在进程线程调度前，我们需要关中断，退出时再恢复中断。接着，我们判断当前可调度的线程数量，如果 `readyProgram` 为空，那么说明当前系统中只有一个线程，因此无需进行调度。

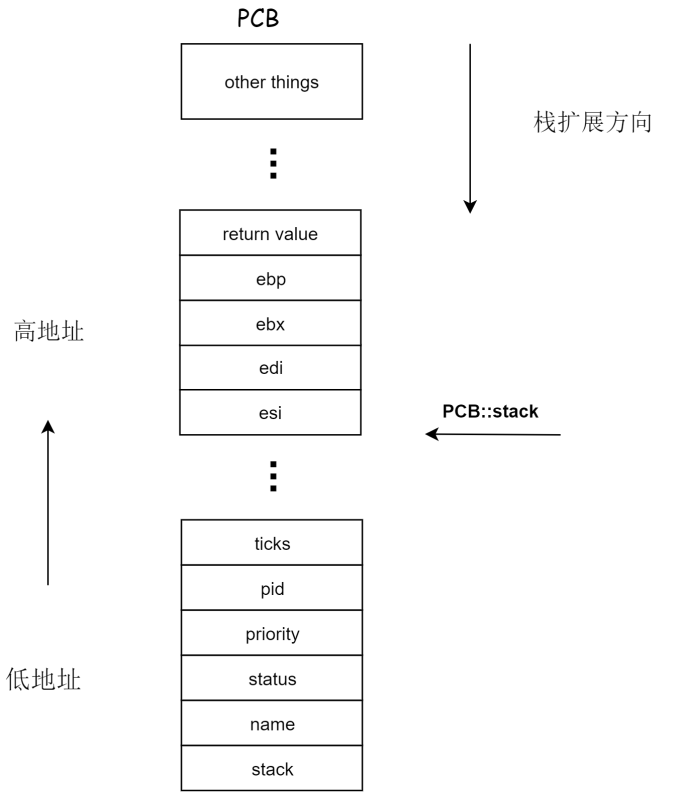
否则，我们判断当前线程的状态，如果是运行态(RUNNING)，则重新初始化其状态为就绪态(READY)和 `ticks` ，并放入就绪队列，其他情况不做处理。然后定义一个指向被换下处理器的线程的指针 `cur` 。接着我们去就绪队列的第一个线程作为下一个执行的线程，从就绪队列中删去这个线程，设置其状态为运行态和当前正在执行的线程。

然后我们就开始将线程从 `cur` 切换到 `next` ，代码如下。线程的所有信息都在线程栈中，只要我们切换线程栈就能够实现线程的切换，线程栈的切换实际上就是将线程的栈指针放到esp中。

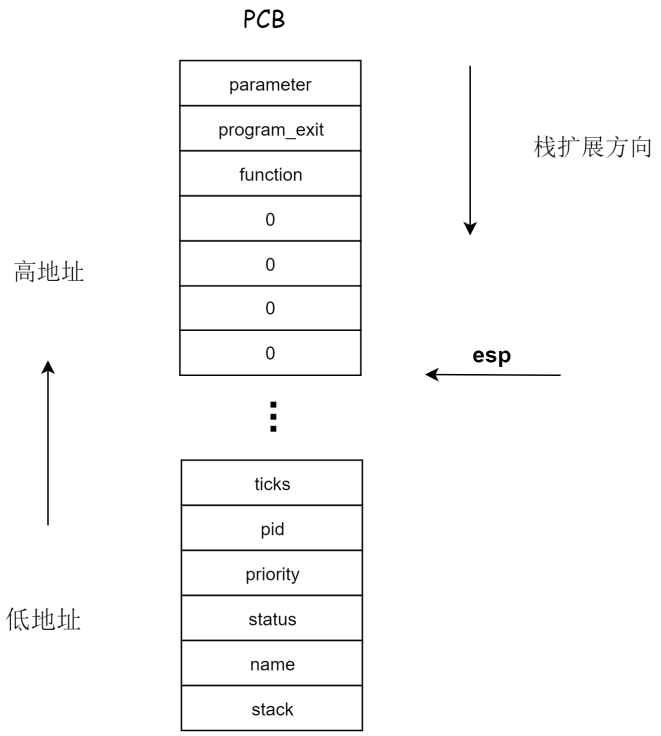
```
1   asm_switch_thread:
2           push ebp
3           push ebx
4           push edi
5           push esi
6
7           mov eax, [esp + 5 * 4]
8           mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
9
10          mov eax, [esp + 6 * 4]
11          mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
12
13          pop esi
14          pop edi
15          pop ebx
16          pop ebp
17
```

```
18        sti
19        ret
```

首先我们保存寄存器 `ebp` ， `ebx` ， `edi` ， `esi` 。为什么要保存这几个寄存器？这是由C语言的规则决定的，C语言要求被调函数主动为主调函数保存这4个寄存器的值。如果我们不遵循这个规则，那么当我们后面线程切换到C语言编写的代码时就会出错。然后，我们保存esp的值到线程的PCB中，用做下次恢复。注意，7-8行代码是首先将 `cur->stack` 的值放到 `eax` 中，然后向 `[eax]` 中写入 `esp` 的值，而 `[eax]` 等于 `*(cur->stack)` ，也就是将esp写入cur指向的线程的栈指针，此时，cur指向的PCB的栈结构如下。



10-11行是将PCB的成员 `stack` 保存的线程栈指针的值写入到esp中，此时， `next` 指向的线程有两种状态，一种是刚创建还未调度运行的，一种是之前被换下处理器现在又被调度。注意，由于esp发生了变化，此时的栈也就跟着发生变化。这两种状态对应的栈结构有些不一致，对于前者，其结构如下。



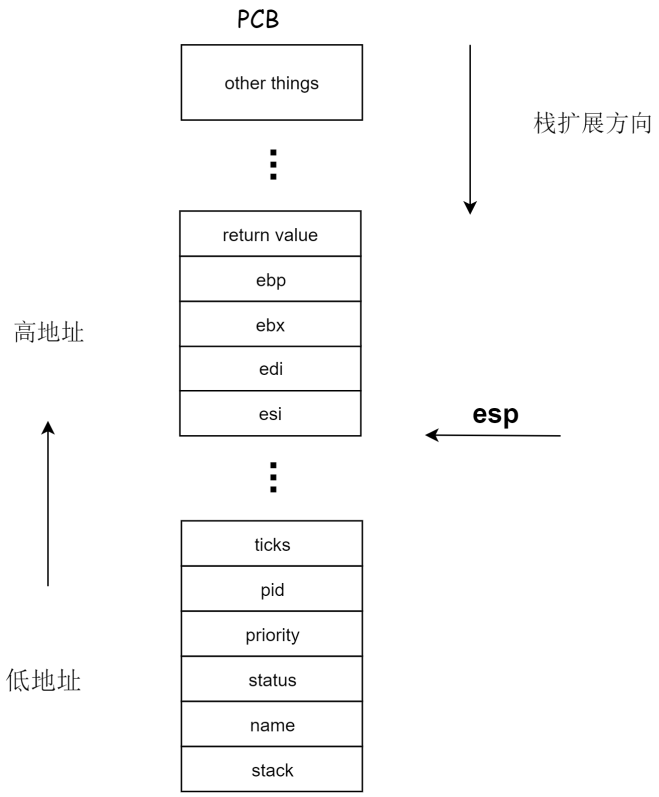
接下来的 `pop` 语句会将4个0值放到esi，edi，ebx，ebp中，此时栈顶的数据是线程需要执行的函数的地址 `function` 。执行ret返回后， `function` 会被加载进eip，从而是CPU跳转到这个函数中执行。此时，进入函数后，函数的栈顶是函数的返回地址，返回地址之上是函数的参数，符合函数的调用规则。而函数执行完成时，其执行ret指令后会跳转到返回地址 `program_exit` ，如下所示。

```
void program_exit()
{
      PCB *thread = programManager.running;
      thread->status = ThreadStatus::DEAD;

      if (thread->pid)
      {
            programManager.schedule();
      }
       else
      {
            interruptManager.disableInterrupt();
            printf("halt\n");
            asm_halt();
      }
}
```

`program_exit` 会将返回的线程的状态置为DEAD，然后调度下一个可执行的线程上处理器。注意，我们规定第一个线程是不可以返回的，这个线程的pid为0。

第二种情况是之前被换下处理器现在又被调度，其栈结构如下所示。



执行4个 `pop` 后，之前保存在线程栈中的内容会被恢复到这4个寄存器中，然后执行ret后会返回调用 `asm_switch_thread` 的函数，也就是 `ProgramManager::schedule` ，然后在 `ProgramManager::schedule` 中恢复中断状态，返回到时钟中断处理函数，最后从时钟中断中返回，恢复到线程被中断的地方继续执行。

这样，通过 `asm_switch_thread` 中的 `ret` 指令和 `esp` 的变化，我们便实现了线程的调度。

> `asm_switch_thread` 的设计比较巧妙，需要同学们结合函数的调用规则，线程栈的设计等知识综合分析。

至此，关于线程的内容我们已经实现完毕，接下来我们来编译运行。

Assignment 3 第一个线程

> assignment 3：创建第一个线程，并输出“Hello World”，pid和线程的name。注意，第一个线程不可以返回。

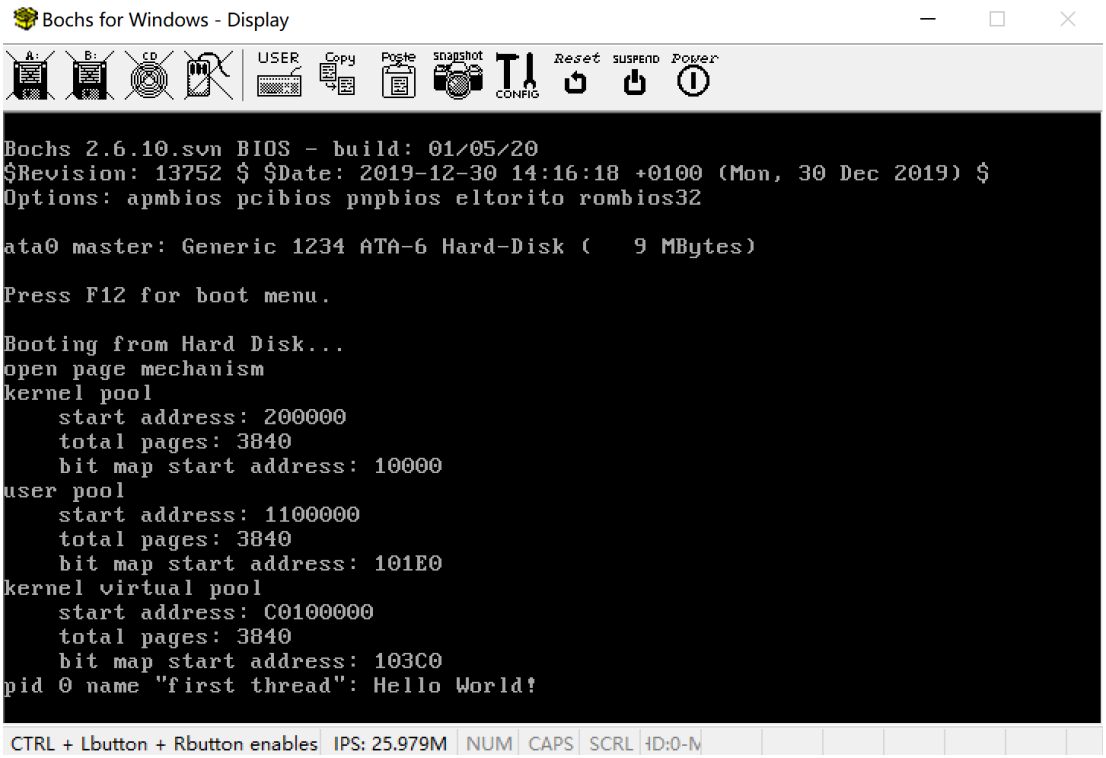
代码在 `src/kernel/setup.cpp` 中，如下所示。

```
1  void first_thread(void *arg)
2  {
3      // 第1个线程不可以返回
4      printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
   programManager.running->name);
5      asm_halt();
6  }
```

```
7
8   extern "C" void setup_kernel()
9   {
10      // 中断管理器
11      interruptManager.initialize();
12      interruptManager.enableTimeInterrupt();
13      interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
14
15      // 输出管理器
16      stdio.initialize();
17
18      // 内存管理器
19      memoryManager.openPageMechanism();
20      memoryManager.initialize(32 * 1024 * 1024);
21
22      // 进程/线程管理器
23      programManager.initialize();
24
25      // 创建第一个线程
26      int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
27      if (pid == -1)
28      {
29          printf("can not execute thread\n");
30          asm_halt();
31      }
32
33      ListItem *item = programManager.readyPrograms.front();
34      PCB *firstThread = programManager.ListItem2PCB(item);
35      firstThread->status = RUNNING;
36      programManager.readyPrograms.pop_front();
37      programManager.running = firstThread;
38      asm_switch_thread(0, firstThread);
39
40      asm_halt();
41  }
```

首先，我们新建一个线程。由于当前系统中没有线程，因此我们无法通过在时钟中断调度的方式将第一个线程换上处理器执行。因此我们的做法是找线程的PCB，然后手动执行类似 `schedule` 的过程，最后执行的 `asm_switch_thread` 会强制将第一个线程换上处理器执行。

最后我们编译运行，输出如下结果。



至此，本章的内容已经讲授完毕。

> 本章的内容刚开始理解起来比较抽象，请同学们结合代码仔细体会😅

练习

1. 请复现assignment 1，并提供完整的测试样例来检查bitmap是否有bug。
2. 请复现assignment 2，并满足如下需求。
    - 抛弃直接指定内存地址的方式，在实模式下使用BIOS中断获取内存的大小，并放置到合适的位置，然后在建立页内存分配时再将其读出。
    - 提供完整的测试样例来检查页内存分配是否有bug。
3. 本章中为什么说“对于任意一个真正的内核虚拟地址 `address`，虚拟地址 `address+0xc0000000` 和 `address` 具有相同的物理地址。”
4. 为什么在 `MemoryManager::initialize` 中要把三个地址池的位图放到0~1MB中，这样做有什么好处。
5. 请用自己的话说说虚拟地址变换到物理地址的三个步骤。
6. 请构造 `0x3423891f` 的页表项和页目录项的虚拟地址，假设第1023个页目录项指向了页目录表。
7. 请问为了构造一个虚拟地址对应的页目录项和页表项的虚拟地址，我们是否设置一个页目录项指向页目录表？这个页目录项的位置是否一定要放在最后一个页目录项中？
8. 请问程序、进程和线程的三者之间的区别是什么？
9. 相比于用户线程，内核线程有什么优势？
10. 请实现根据 `List` 的声明实现你自己的 `List`。
11. 请复现Assignment 3。

12. 请多创建几个线程，演示多线程并发的效果。

bonus

1. 请对 `allocatePid` 做出改进。
2. 请实现其他的线程调度算法，并比较它们之间的异同。