COA2020

1.实验要求

1.1 完成实模式下的地址转换

Method In MMU

```
  /**
   * 实模式下的逻辑地址转线性地址
   * @param logicAddr 48位 = 16位段选择符(高13位index选择段表项) + 32位offset，计算公式为：①(16-bits段寄存器左移4位 + offset的低16-bits) = 20-bits物理地址 ②高位补0到32-bits
   * @return 32-bits实模式线性地址
   */
  private String toRealLinearAddr(String logicAddr){
    //TODO
    return null;
  }
```

Method In Memory

```
    /**
     * 实模式下从磁盘中加载数据
     * @param eip 实模式下，内存0-32MB地址对应磁盘0-32MB地址
     * @param len 数据段长度
     */
    public void real_load(String eip, int len){
        //TODO
        char[] data = null;
        write( eip,len, data );
    }
```

1.2 完成段模式下的地址转换

Method In MMU

```
  /**
   * 根据逻辑地址找到对应的段号
   * 段选择符高13-bits表示段描述符的索引，低3-bits本作业不使用
   * @param logicAddr 逻辑地址
   * @return
   */
  private int getSegIndex(String logicAddr){
```

```
    //TODO
    return -1;
  }
  /**
   * 分段模式下的逻辑地址转线性地址
   * @param logicAddr 48位 = 16位段选择符(高13位index选择段表项) + 32位段内偏移
   * @return 32-bits 线性地址
   */
  private String toSegLinearAddr(String logicAddr) {
    //TODO linearAddr = SegBase + offset
    return null;
  }
```

完善Read方法

Method in Memory

```
    /**
     * 段式存储模式下，从磁盘中加载数据.段页式存储中，不用从磁盘中加载数据
     * @param segIndex 段号
     * @return 该段的内存基址
     */
    public String seg_load(int segIndex) {
        //TODO
        SegDescriptor targetSeg = segTbl.get( segIndex );


        //TODO 使用 removeSegByLRU 和 occupiedSize 在内存中空出一块足够且连续的空间加
载段
        // code here


        //TODO 使用 fit 从磁盘中加载数据
        // 段页式时不需要从磁盘中加载数据
        if(Memory.PAGE){
            // Code here
            // You don't need to load data from disk
        }else{
            // Code here
        }

        return String.valueOf(  targetSeg.getBase());
    }
```

1.3 完成段页式下的地址转换

Method In MMU

完善Read方法

Method In Memory

```
    /**
     * 段页式存储下，从磁盘中加载数据
     * @param segIndex 段号
     * @param pageNO 虚拟页号
     * @return 该页的起始地址32-bits
     */
    public String page_load(int segIndex, int pageNO) {
        //TODO
        // 首先，你需要把页号对应的数据加载到内存中
        // 第二，你需要从段中表达的那些页框里选出一个和虚页号对应起来，如果全都映射到了虚页，那你需要使用我们提供的removePageByLRU来获取一个有效页
        // 第三，设置好反向页表和页表
        // 最后，添加TLB，在往TLB里放入虚页：页框项的时候，需要把数据也加载到内存之中，否则会出现TLB命中，页表命中，Cache访问内存物理地址，但是内存却没有这个数据的尴尬情况。
        return null;
    }
```

1.4 融合Cache与TLB

Method In MMU

完善Read方法

2.相关资料

Virtual Memory

为了允许多个程序共享相同的内存，我们必须能够保护不同的程序，确保程序只能读和写分配给它的主存部分。主存只需要包含许多程序的活动部分，就像$Cache$只包含一个程序的活动部分一样。局部性对虚拟内存和缓存来说都适用，而虚拟内存允许我们高效地共享处理器和主存。

当我们编译程序时，我们无法知道哪些程序将与其他程序共享内存。实际上，在程序运行时，共享内存的程序会动态更改。由于这种动态交互，我们希望将每个程序编译成它自己的地址空间——一个仅供该程序访问的内存位置的单独范围。虚拟内存实现了程序地址空间到物理地址的转换。这个转换过程加强了对其他程序的程序地址空间的保护。我们可以写一个最简单的Hello World来体会上面这段话（后面所有的代码执行环境都是Linux）

```
## vim a.c
include<stdio.h>
int main(){
        printf("Hello World");
        return 0;
}
```

然后在Terminal中输入

```
gcc a.c -o a.out && gcc a.c -o b.out
diff a.out b.out
```

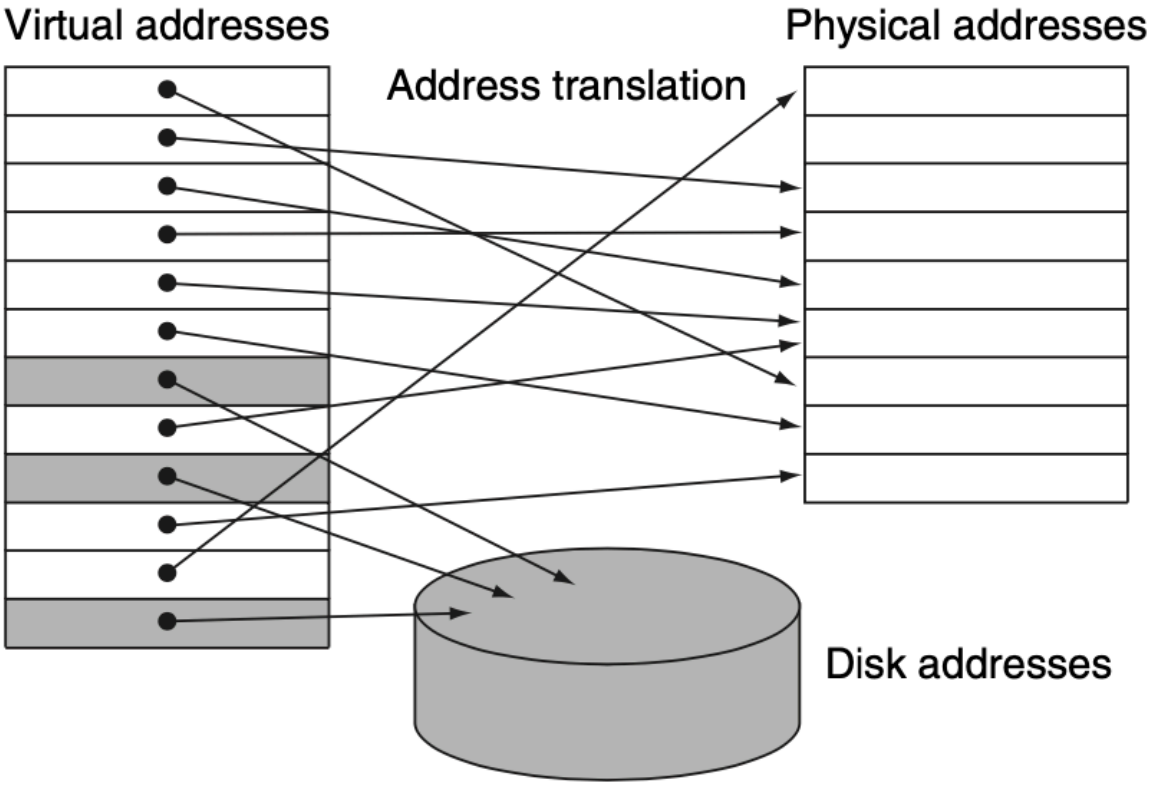
如果我们使用`objdump -d a.out`可以发现反汇编出来的代码中不同的函数已经被标上了不同的地址。大家可以发现两个程序编译出来的可执行文件拥有完全相同的程序地址空间。

> Extension：gcc是一个编译C语言的工具，我们都知道编译C程序有四个步骤：预处理->编译->汇编->链接，这四件事都是由gcc一个人完成的吗？Strace可以跟踪到一个进程产生的系统调用,包括参数，返回值，执行消耗的时间。利用Strace，我们就可以看到整个gcc工作的流程，But how to ...？

另一方面，使用虚拟内存的原因是我们希望允许单个用户程序超过主内存的大小。上个世纪，如果一个程序太大，内存不够用，则由程序员来完成这个从硬盘加载到内存的步骤。程序员将程序分割成小块，然后识别出互不相容的小块。这些块在执行期间由用户程序控制加载或卸载，程序员要确保程序不会试图访问未加载的块，并且加载的块不会超过内存的总大小。块在传统上被组织为模块，每个模块都包含代码和数据。不同模块中的过程之间的调用将导致一个模块与另一个模块重叠。

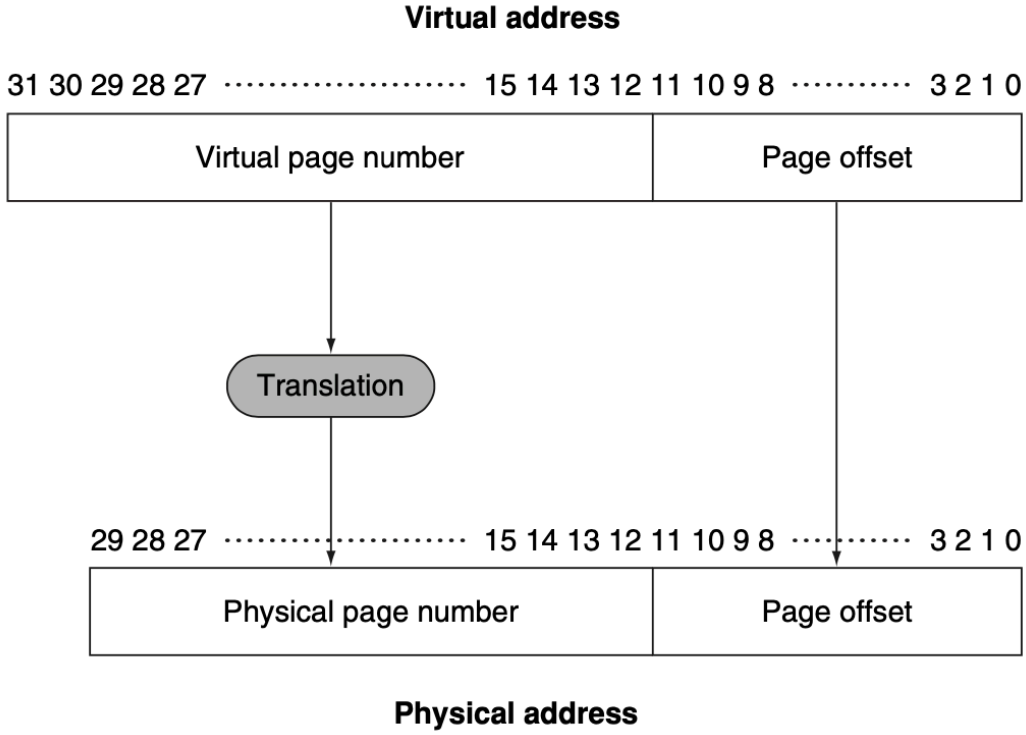
可以想象，这种责任对大家来说是一个巨大的负担（笑）。发明虚拟内存就是为了让程序员摆脱这一困难，它可以自动管理由主存和辅助存储表示的内存层次结构的两个级别。

虽然在虚拟内存和缓存中工作的概念是相同的，但它们不同的历史根源导致使用不同的术语。虚拟内存块称为页面，虚拟内存丢失称为页面错误。在虚拟内存中，处理器产生一个虚拟地址，这个地址由硬件和软件的组合转换成一个物理地址，而物理地址又可以用来访问主存。



上图显示了虚拟地址和物理地址转换的过程。这个过程称为地址映射或地址转换。如果和我们的图书馆类比，我们可以认为一个虚拟地址作为一本书的标题，一个物理地址作为那本书在图书馆的位置。

虚拟内存还通过提供重定位来简化程序的加载。重定位将程序使用的虚拟地址映射到不同的物理地址，然后这些地址才被用来访问内存。这种重新定位允许我们在主存的任何位置加载程序。此外，现在使用的所有虚拟内存系统都将程序重新定位为一组固定大小的页面，因此不需要找到相邻的内存块来分配给程序，操作系统只需要在主存中找到足够数量的页面。



在虚拟内存中，地址被分解为虚拟页码和页偏移量。物理页号构成物理地址的上部，而未改变的页偏移量构成下部。页偏移量字段中的位数决定了页的大小。可通过虚拟地址寻址的页面数量不需要与可通过物理地址寻址的页面数量匹配。拥有比物理页面更多的虚拟页面是虚拟内存无限大的假设的基础。

存储管理

在IA-32下，CPU主要有两种工作模式，在本次作业中对应着两种地址转换的模式。分别是实模式和保护模式

实模式

在实模式下，虚拟地址就是物理地址，我们直接进行寻址，这也导致了我们的程序无法访问超过内存容量的那些数据，比如4GB的内存，我们就只能访问磁盘的前4GB。在实模式下，我们对整个内存空间进行了分段，段在物理上是连续的，且程序装入内存的时候需要按照指定的位置设施代码段，数据段，栈段的寄存器。这种实模式具有一定的缺点

- 安全性问题
    - 程序采用物理地址来实现访存，无法实现对程序的代码和数据的保护
    - 一个程序可以通过改变段寄存器和偏移寄存器访问并修改不属于自己的代码和数据
- 分段机制本身的问题
    - 段必须是连续的，从而无法利用零碎的空间
    - 段的大小有限制(最大为 64KB)，从而限制了代码的规模

保护模式

这里的段式不同于实模式中的段式，实模式中的段是固定静止的，在段式中的分段机制是使用一个段选择子来寻址。具体是这样的，我们会有一个全局描述符表（GDT），在保护模式寻址的时候，我们会从地址中计算出一个段选择子（Selector），并根据Selector在GDT中找到一个对应的段描述符。这个段描述符中包含了段的真正物理首地址，我们用这个物理地址和偏移量就可以正确寻址了

Selector一般来说是16位，高13位表示他在GDT中的索引，低3位表示权限等信息。GDT由一个个段描述符组成，段描述符定义了段的起始32位物理地址，段的界限，属性等内容。

反向页表

计算机的逻辑地址空间越来越大，页表所占用的内存空间也越来越多，页表尺寸与虚拟地址空间成正比增长，除了使用多级页表来解决这个问题以外，也有的操作系统使用反向页表来解决这个问题。他的优点是所有进程只需要维护一张表，这个表为内存中的每个页框都建立一个表项，并且按照块号进行排序。关于反向页表，在实验攻略中有更详细的描述。

3.实验攻略

因为平台给每位同学分配的内存大小是有限的，所以我们使用了去年磁盘文件，对同学们正常编码应该不会造成影响。写之前建议先阅读一遍已经给出的代码，理解一下代码框架对大家编码有好处。

数据结构

我们已经给大家写好了一个TLB，它的结构和Cache类似，在这里就不做赘述，本次实验数据结构的重点在Memory中。我们在Memory中给大家实现好了一个页表，其中页表项的数据结构是这样的

```
    /**
     * 页表项为长度为20-bits的页框号
     */
    private class PageItem {

        private char[] frameAddr;

        private boolean isInMem = false;

        public char[] getFrameAddr() {
            return frameAddr;
        }

        public void setFrameAddr(char[] frameAddr) {
            this.frameAddr = frameAddr;
        }

        public boolean isInMem() {
            return isInMem;
        }

        public void setInMem(boolean inMem) {
            isInMem = inMem;
        }

    }
```

其中frameAddr表示的是这个虚页对应物理页号。我们也为大家提供了一个段描述符的数据结构，相应的注释已经写在代码中了。段描述符被存放在一个叫做`segTbl`（全局描述符表）的数据结构中。大家需要根据段号在里面存取。

除此之外，我们还为大家提供了一个反向页表。页表存储的是虚页号与页框号的对应关系，反向页表就是将页表反了过来，存储的是页框号与虚页号的对应关系。在框架代码中，我们是在MMU中通过内存或者TLB获取页框号，再将页框号传入Memory中获取实际的数据。我们在Memory.read中使用的是反向页表来读数据。

```
private static ReversedPageItem[] reversedPageTbl = new
ReversedPageItem[Memory.MEM_SIZE_B / Memory.PAGE_SIZE_B]; // 反向页表大小为2^15
32K

/**
     * 反向页表 32 * 1024 * 1024 / 1 * 1024 = 32 * 1024 = 32 KB 个反向页表项
     */
```

```
    private class ReversedPageItem {

        private boolean isValid = false;    // false表示不在内存，true表示在内存中

        private int vPageNO = -1;           // 虚页页号

        private long timeStamp = System.currentTimeMillis();

        public long getTimeStamp(){
            return this.timeStamp;
        }

        public void updateTimeStamp(){
            this.timeStamp = System.currentTimeMillis();
        }

    }
```

代码导读

```
    /**
     * 最先适应算法
     * @param index 要考虑的segDescriptor
     * @param data 要存入的数据
     * @param len data的长度
     * @return
     */
    public void fit(int index, char[] data, int len) {
        List<AbstractMap.SimpleEntry<Integer, Integer>> occupiedSegs = new
ArrayList<>(  );   // 存储每个被占用的内存空间的segDescriptor的base与index

        // 下面这个循环会计算出目前所有有效段占用的空间
        for(SegDescriptor segDes: segTbl){
            if(segDes.isValidBit()){    // 已经被占用
                int base = chars2int( segDes.getBase() );
        AbstractMap.SimpleEntry<Integer, Integer> pair = new
AbstractMap.SimpleEntry<>( base, segTbl.indexOf( segDes ) );
                occupiedSegs.add( pair );
            }
        }

        Collections.sort( occupiedSegs, new
Comparator<AbstractMap.SimpleEntry<Integer, Integer>>() {  // 按起始位置从小到大排
序
            @Override
            public int compare(AbstractMap.SimpleEntry<Integer, Integer> p1,
AbstractMap.SimpleEntry<Integer, Integer> p2) {
                return p1.getKey()-p2.getKey();
```

```
            }
        } );

        int current = 0;    // 可以看成一个内存指针
        int totalFree;   // 已经找到的空闲空间
//        System.out.println("Start Fit:" + occupiedSegs.size());
        for(AbstractMap.SimpleEntry<Integer,Integer> occupied: occupiedSegs){
            int base = occupied.getKey();
            // 段基址相对于当前指针的偏移
            totalFree = base-current;
            if(totalFree >= len) {
                break;
            }
            SegDescriptor segDes = segTbl.get( occupied.getValue() );
            int limit = chars2int( segDes.getLimit());
            if(occupied.getKey()!=current){    // 整理碎片
                //读出段中的数据
                char[] segData = read_for_manage( String.valueOf(
segDes.getBase() ), limit);
                //整体前移
                write( String.valueOf( int2chars( current ) ), limit, segData
);
                //重写段基址
                segDes.setBase( t.intToBinary( String.valueOf( current
)).toCharArray());
            }

            current += limit;

        }

        SegDescriptor targetSegDes = segTbl.get( index );
        targetSegDes.setBase( int2chars( current ) );
        targetSegDes.setValidBit( true );
        targetSegDes.updateTimeStamp();
        // 分段模式下载入内存，段页模式下不载入内存
        if(!Memory.PAGE){
            write( String.valueOf( int2chars( current )) , chars2int(
targetSegDes.getLimit()) ,data);
        }

    }
```

这段代码实现了内存中段的移动和设置新的段，这个算法没有考虑数据无法塞入内存的情况，因为在使用之前应该已经确保内存中一定可以整理出一段放入data的空间。

实现指导

实模式

略

段模式

从逻辑地址中拿到段选择子，然后根据段选择子拿到基址，接着根据偏移量寻址即可，其实段式的地址转换就是上文保护模式下寻址的方式，只不过作业中有所简化。段在磁盘中是连续存储的，且一旦初始化，其磁盘基址和限长就被固定不会发生变化(内存基址可能改变)。

段页式

段页式其实就是将每个段分成了不同的页来管理，现在我们的内存空间被分成了几个段，每个段里面又被分成了好几个页，这些页其实就是对内存所进行的分页。假如我们有一个10KB大小的段，段的基址是`0x00000000`，在段页式中，这个段表达了10个页（如果页的大小是1KB的话），并且这10个页恰好是内存从`0x00000000`开始的十个页。这些页其实就是可能成为页框的页。所以段页式下寻址时我们需要先把段加载了，加载页的时候，我们需要查看段中的哪些页是可以用的，因为有的页已经和虚页映射起来了，这些页我们不能使用。

在加载页的时候，大家不仅需要把页表中对应的项修改成功，还需要修改一下反向页表，修改的方式很简单。因为段中的页和内存中的页是一一对应的，所以你在段中获取的页号就是你需要修改的反向页表的页号，这样就可以取出对应的反向页表项来更新了。

TLB与Cache

虽然TLB已经写好了，但是把TLB融合到我们的代码中还需要自己多做一些事情，大家一定要小心操作，不然会发生很多奇怪的错误。任何修改了页表、反向页表的地方都是大家应该谨慎的地方，因为页表和反向页表的修改极有可能导致TLB的修改，这些地方可能都需要大家添加代码。在加载页同时加载TLB的时候，具体该加载多少页进入TLB呢？我们建议大家加载从当前页开始往后半个TLB容量的页，这样不至于访问磁盘的时间过长，也不至于TLB的命中率过低。当然这些数据是根据测试用例统计的结果。关于加载的方式和容量大家可以自己决定，这里只给出一个建议。

将Cache融合到代码中比较简单，这里不过多赘述。

测试

我们的测试分成了实模式、段模式、段页式和TLB四个，每一个种我们都提供一个用例给大家参考（除了TLB，因为他只有一个用例，但是他和段页式是一样的）

```
// 段页式
    @Test
    public void test1() {
        String eip = "00000000000000000000000000000000";
        int len = 2 * 1024;
        char[] data = helper.fillData((char) 0b00001111, len);
        memory.alloc_seg_force(0, eip, len / 2, true, "");
        assertArrayEquals(data,
mmu.read("00000000000000000000000000000000000000000000000000000000", len));
```

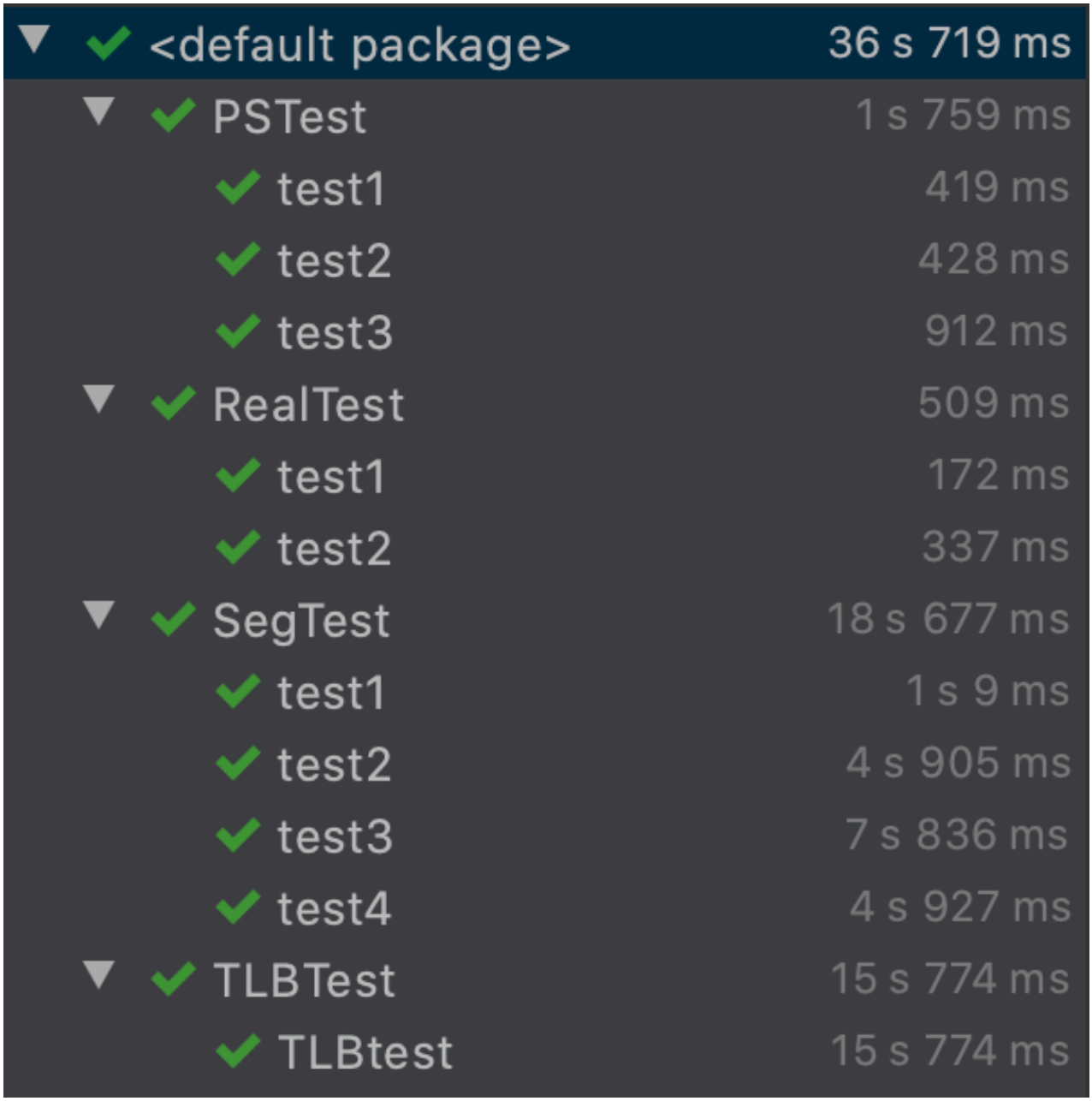
```
    }
// 实模式
  @Test
  public void test1() {
    int len = 128;
    char[] data = helper.fillData((char)0b00001111, 128);
    assertArrayEquals(data,
mmu.read("00000000000000000000000000000000000000000000000000", len));
  }
// 段式
  @Test
  public void test1() {
    String eip = "00000000000000000000000000000000";
    int len = 1024 * 1024;
    char[] data = helper.fillData((char)0b00001111, len);
    memory.alloc_seg_force(0, eip, len, false, eip);
    assertArrayEquals(data,
mmu.read("00000000000000000000000000000000000000000000000000", len));
  }

  public char[] fillData(char dataUnit, int len) {
    char[] data = new char[len];
    Arrays.fill(data, dataUnit);
    return data;
  }
```

注意，这些用例仅输入输出和平台上是一样的，但是不代表我们在平台上也使用了assertArrayEquals来比较大家的结果。另外TLB的用例超时会导致大家其他用例都通过却只有0分，所以我们通过TLB里面的`isAvaliable`这个属性来判断是否执行这个用例，建议大家先把这个字段设置成`False`，这样这个用例会直接判错，但是不会超时，大家也可以拿到其他用例的分数，等大家把TLB融合到代码里之后再打开这个字段。

提示

读取磁盘是一件非常慢的事情，所以大家尽量少使用disk.read函数，读一次磁盘大概有150ms的延迟。同时我们有一个用例会不断的发生缺页，因此会不断的读取磁盘，如果大家不实现一个TLB或者TLB每次装载的页数太少的话，会导致1分钟以内无法执行完我们的用例，大家可以自己修改TLB类中的参数让TLB更大，但是尽量不要太离谱了。以下是我在自己的电脑上执行用例的结果（添加了TLB，没添加TLB的时候五分钟也没有执行完，我的电脑是i7 6核 32GB内存）。另外，框架代码只能在有限范围内是正确的，如果大家发现了什么Bug或者有什么修改意见，非常欢迎写邮件给任老师！



4.代码框架

```
.
├── pom.xml
└── src
    └── main
        └── java
            ├── cpu
            │   ├── MMU.java                    # you need write
            │   ├── TLB.java
            │   └── alu
            │       ├── ALU.java
            │       ├── FPU.java
            │       └── NBCDU.java
            ├── memory
            │   ├── Cache.java
```

```
│   ├── Disk.java
│   ├── DiskInterface.java
│   ├── Memory.java                          # you need write
│   ├── MemoryInterface.java
│   ├── cacheMappingStrategy
│   │   ├── AssociativeMapping.java
│   │   ├── DirectMapping.java
│   │   ├── MappingStrategy.java
│   │   └── SetAssociativeMapping.java
│   ├── cacheReplacementStrategy
│   │   ├── FIFOReplacement.java
│   │   ├── LFUReplacement.java
│   │   ├── LRUReplacement.java
│   │   └── ReplacementStrategy.java
│   └── cacheWriteStrategy
│       ├── WriteBackStrategy.java
│       ├── WriteStrategy.java
│       └── WriteThroughStrategy.java
├── transformer
│   └── Transformer.java
└── util
    ├── BinaryIntegers.java
    ├── CRC.java
    └── IEEE754Float.java
```

5.未尽事宜

请发邮件给任老师