# 作业5

本次作业共包含3道大题，总计25分。

---

## Q1 页表（16分）

已知内存是字节可寻址的、每次内存访问针对的是32-bit的word、虚拟地址24位、物理地址20位、页面大小为4096字节、TLB是二路组相联，共有16个TLB项（即2-way set associative with 16 total entries）。在下面的表格中，所有的数字都是十六进制的。TLB和页表前32项的内容如下：

| TLB | | | |
|---|---|---|---|
| Index | Tag | PPN | Valid |
| 0 | 011 | 1C | 1 |
| | 02C | B3 | 1 |
| 1 | 13C | A4 | 0 |
| | 0B2 | 7E | 1 |
| 2 | 001 | 05 | 1 |
| | 1A3 | B6 | 0 |
| 3 | 002 | 08 | 1 |
| | 003 | 17 | 1 |
| 4 | 1C2 | 21 | 1 |
| | 013 | 09 | 0 |
| 5 | 1CF | 38 | 1 |
| | 08B | 51 | 1 |
| 6 | 003 | 7A | 1 |
| | 13C | 7F | 1 |
| 7 | 031 | B2 | 0 |
| | 0A4 | 3C | 1 |

| Page Table | | | | | |
|---|---|---|---|---|---|
| VPN | PPN | Valid | VPN | PPN | Valid |
| 00 | 04 | 1 | 10 | 03 | 1 |
| 01 | 13 | 0 | 11 | 24 | 0 |
| 02 | 28 | 1 | 12 | 1E | 1 |
| 03 | 1F | 1 | 13 | 08 | 1 |
| 04 | 3E | 1 | 14 | 02 | 1 |
| 05 | 6C | 0 | 15 | 2A | 1 |
| 06 | 09 | 1 | 16 | 3C | 0 |
| 07 | 17 | 0 | 17 | 3B | 0 |
| 08 | 24 | 1 | 18 | 1C | 1 |
| 09 | 07 | 1 | 19 | 16 | 0 |
| 0A | 05 | 1 | 1A | 21 | 0 |
| 0B | 2D | 1 | 1B | 17 | 1 |
| 0C | 06 | 0 | 1C | 22 | 1 |
| 0D | 3B | 1 | 1D | 2E | 0 |
| 0E | 21 | 1 | 1E | 7A | 1 |
| 0F | 08 | 0 | 1F | 4B | 1 |

### Q1.1 地址格式（2分）

在图上标注出虚拟、物理地址中所包含的字段（下面所列举的字段若存在则标记，若不存在跳过即可）：

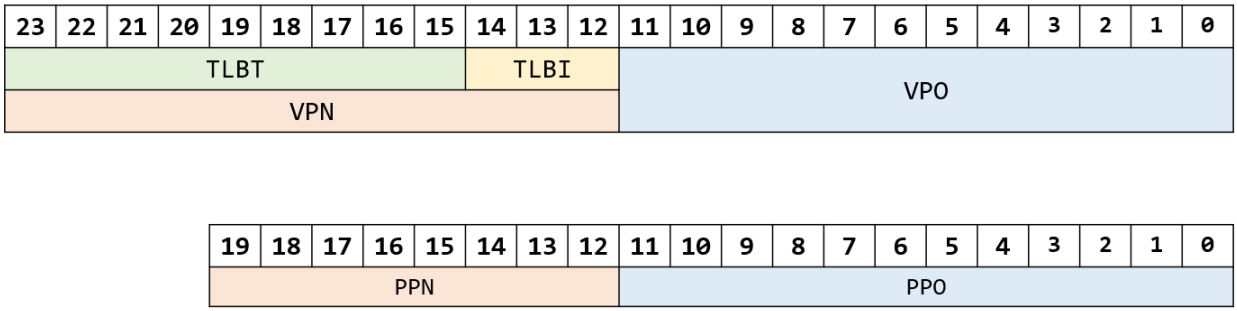
---

> 参考课件《7-虚存》P.20 的基本参数和符号。

约定虚拟地址为`vaddr[]`、物理地址为`paddr[]`，使用Python切片来描述：如`vaddr[1:12]`表示虚拟地址的第`1`位至第`11`位（从`0`开始，区间左闭右开）、`vaddr[12:]`表示虚拟地址第`12`位至最后一位。

由$P = 4096 = 1024 \times 4 = 2^{12} \Rightarrow p = 12$可知`VPO`与`PPO`的位宽为12位，于是有`VPO = vaddr[0:12]`、`VPN = vaddr[12:] = vaddr[12:]` `PPO = paddr[0:12]`，`PPN = paddr[12:]`。

使用`VPN`去查询TLB，因此`TLBI`和`TLBT`为`VPN`的子域。从上表中可知 TLB 一共有 8 个 set，index 为`[0, 8)`，每个 set 为二路组相连：`TLBI`为3位，即`TLBI = VPN[0:3] = vaddr[12:15]`；余下为`TLBT = VPN[3:] = vaddr[15:]`。







## Q1.2 虚拟地址转换（14分，一项1分）

对于给定的两个虚拟地址 `0x01DBE3` 、 `0x9E6CF2` ，请分别写出 相应的TLB 表项和物理地址，并指出：

- TLB 是否命中（回答 `Y` 或 `N` ）
- 是否发生 page fault（回答 `Y` 或 `N` ）
  - 以页表前 32 项为准；即如果不在页表前 32 项内，则发生page fault；系统保证page fault异常处理后可以解决问题
  - 如果发生 page fault，请在 `PPN` 一项填 `-` 。
- 获得该地址所存储数据的访问时间（以下简称 "访问时间"）：
  - 假设一次内存访问时间100ns，一次快表（TLB）访问时间为10ns，处理一次缺页需要108 ns （含更新TLB和页表的时间）

---

> 参考课件《7-虚存》P.28~33的示例。

假定所有的数据保存在内存中，对照上面的地址格式，写出虚拟地址对应的域后依次查TLB和页表，并记录从获取有效物理地址到**取出实际数据**的实际用时：

- `0x01DBE3`：
  i. `VPO = 0xBE3` , `VPN = 0x1D` , `TLBI = 0x5` , `TLBT = 0x003`
  ii. 查询 `index = 5` 的 TLB set，不包含 TLB tag `0x003` ，触发 TLB Miss：用时 10 ns
  iii. 按 `VPN = 0x1D` 查询页表， `Valid` 位为 `0` ，触发 Page Fault（见课件《7-虚存》P.23）：用时 100 ns
  iv. 处理 Page Fault，更新页表与TLB：用时 108 ns **（见下面勘误）**
  v. 再次访问TLB，TLB Hit：用时 10 ns
  vi. 获得物理地址，访问内存取出数据：用时 100 ns
  总共用时：10 + 100 + 108 + 10 + 100 = 328 ns
- `0x9E6CF2`：
  i. `VPO = 0xCF2` , `VPN = 0x9E6` , `TLBI = 0x6` , `TLBT = 0x13C`
  ii. 查询 `index = 6` 的 TLB set，包含 TLB tag `0x13C` ，TLB Hit： `PPN = 0x7F` ，用时 10 ns
  iii. 获得物理地址，访问内存取出数据：用时 100 ns
  总共用时：10 + 100 = 110 ns

| | `0x01DBE3` | `0x9E6CF2` |
|---|---|---|
| VPN | `0x01D` | `0x9E6` |
| TLB Index | `0x5` | `0x6` |
| TLB Tag | `0x003` | `0x13C` |
| TLB Hit? (Y/N) | N | Y |
| Page Fault? (Y/N) | Y | N |
| PPN | `-` | `0x7F` |
| 访问时间（单位为ns） | 328 | 110 |

> ⚠ **勘误：实际缺页处理用时**
>
> 缺页处理需要向外存发出请求，取出新的页表加载到内存中，显然 外存访存速度 $<<$ 内存访存速度（回顾课件《3.1-C语言与汇编》P.7）。
> 本次作业出题时出现了笔误：处理缺页用时实际为 $10^8$ ns，尽管不影响作答，严谨起见在此特别纠正。

## Q2 重定位（3分）

`foo.c` 的内容如下所示，其中 `test_call()` 共执行5次调用，依次为 `test1` ~ `test5`：

```c
// foo.c
extern void (*test1)();

static void test2() {}

void test3() {}

extern void test4();

void test_call(void (*test5)()) {
    test1();
    test2();
    test3();
    test4();
    test5();
}
```

现生成 `foo.o` 文件，在链接期间需要全局重定位的调用有哪些（即生成的 `foo.o` 文件中，需要重定位的符号有哪些）？

---

在链接阶段需要全局重定位的：**被引用的** 全局变量、全局函数入口地址（见课件《6.1 汇编与C语言-5》P.25小结）。

所以需要全局重定位的调用为 `test1`，`test3`，`test4`。

## Q3 `fork()` （6分，每题2分）

阅读程序，并回答问题（每题2分）：

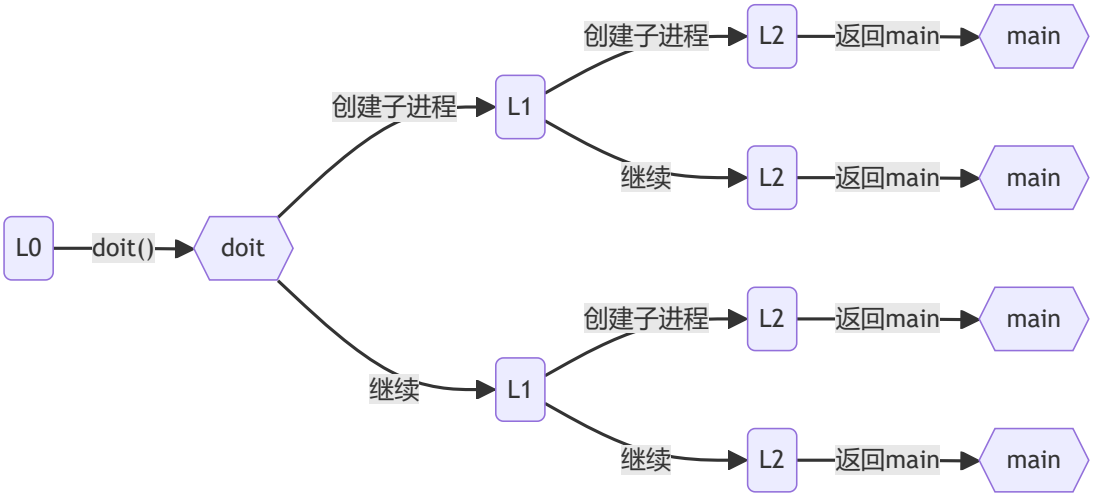
1. 该程序共输出多少行 `hello`？

```c
void doit() {
    fork(); // L0 -> L1, L1
    // L1 从这开始
    fork(); // L1 -> L2, L2，注意有两个L1，所以有两个L2
    // L2 从这开始
    printf("hello\n"); // 1次
    return;
}

int main() {
    doit(); // L0
    // L2 从doit()返回后继续执行
    printf("hello\n"); // 2次
    exit(0);
}
```

---

画出父子进程关系：




所有L2在`doit()`中输出一行`hello`后返回（各进程中的）`main()`再输出一行`hello`，**总共8行`hello`**。

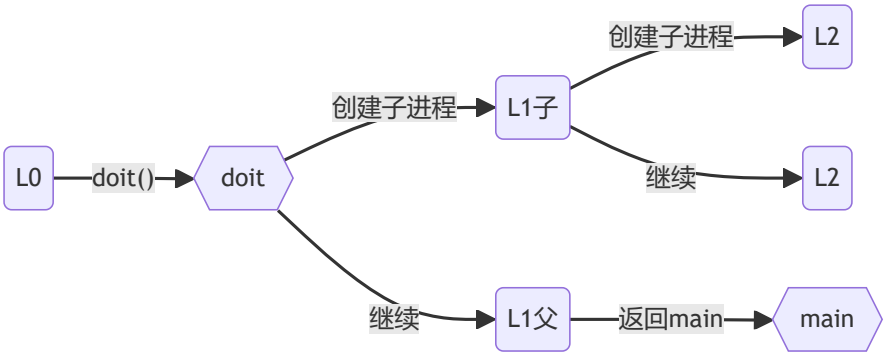
2. 该程序共输出多少行`hello`？

```c
void doit() {
    if (fork() == 0) { // L0 -> L1(子), L1(父), L1(子) 继续
        fork(); // L1 -> L2, L2
        printf("hello\n");
        exit(0);
    }
    return; // L1(父) 从这里返回
}

int main() {
    doit();
    printf("hello\n");
    exit(0);
}
```

---

画出父子进程关系：




`doit()` 里第一次 `fork()` 后，父进程（L1父）直接返回 `main()`；子进程（L1子）继续 `fork()` L2，它们输出后立即 `exit(0);` 而不会返回 `main()`，**总共3行 `hello`**。

3. 该程序输出的 `counter` 值应为多少?

```
int counter = 1; // 从相同的状态开始，但各自具有私有副本
int main() {
    if (fork() == 0) {
        counter--;
        exit(0);
    } else {
        wait(NULL);
        counter++;
        printf("counter = %d\n", counter);
    }
    exit(0);
}
```

---

考察点在于 `fork()` 了之后，尽管从相同状态开始，父子进程实际各自具有私有副本（见课件《9.1-异常控制流》P.27~P.30）。

即，子进程修改自身的 `counter`，不会影响父进程的 `counter`。在这里只有父进程会输出（自身的） `counter` 的值，所以 **输出的 `counter` 的值为 `2`**。