

电子作业

第一次作业：为什么要上大学？

进入大学前，甚至刚上大学的时候，对这个问题是没有什么理解的：毕竟辛苦从小上学上到大，读完了高中，成绩还挺不错，自然就要上大学，没有什么为什么——就像人的出生和死亡，没有原因，就是一种客观、纯粹的行为。

但是到现在也算是在大学里待了两三年，算得上是对大学“祛魅”了，这里有学术研究、为人类贡献的神圣，也有零零碎碎、鸡毛蒜皮，和社会无异。也许还是不能准确的说为什么要来上大学，但是从被大学塑造的“我”这个个体的角度，可以反推出上大学的原因。

首先是获取一份学历，或是本科就业的学士学位、也或是继续深造的到的硕博学位，但总体上是学校为我这个作为劳动力的个体向社会出示的证明。

但更重要的，是进行一些“社会化”。大学的确不是象牙塔，今后在社会上安身立命所需的各种能力、需要的各种认知都在学校里或许明显、或许隐晦的投射在校园生活里，生活上的衣食住行，“能力是最不重要的”社会规则等等。

第二次作业：大学最应该培养和获得的能力

1. 具备与他人合作、共同学习的能力
2. 具有某一领域知识的深度
3. 具备批判性、系统性的思维能力

从做题家到社会人，首先要在思维上建立起评价体系的转换。过去，只要做好自己的这一份试卷，就是个人的成功；而在社会上基本没有这种明显、清晰的评价体系，更多的是要在多人协作的工程、项目里操劳。诚然不会像完成小组作业那样朴素的“一荣俱荣”合作方法，但是也要培养出合作能力，清晰思维，降低负担。

具有某一领域知识深度，纵使有着“能力是最不重要”的潜规则，但对我而言，能力的培养是最具性价比的。好好专精自己的专业领域，有了绝对的能力，再去考虑结合相对意义上的别的能力，比如交往、人情等。

具备批判性、系统性思维能力。实际生活是复杂的。

第三次作业：生活中的并发与并行

并发：厨房里只有一个人做饭，先烧水、再下面、接着切菜.....同一时刻只有一件事在被做，多个任务是逻辑上的同时进行。

并行：三个人一起做饭，一个烧水一个下面一个切菜.....同一时刻，多件事在被做，是物理上的同时进行。

因此，最大的区别就是“是否是真正意义上的同时进行”。

第四次作业：keyboard->OS & OS->monitor

keyboard->OS：首先是硬件中断机制，当按下按键时，键盘控制器会生成一个中断信号，让CPU进行对应的处理，这就是中断处理机制，需要提前注册好的中断处理程序，通常是更改对应映射区域的内存，接着就是扫描码转换为字符，从对应的内存区域获取扫描码、转换为字符，最后放进输入缓冲区。

OS->monitor：首先要对输出数据进行抽象，把图像/文本数据按照定义好的API转换为二进制数据，接着调用写入显存驱动，接着应用程序请求显示，OS负责调度，最后显卡获取像素数据，驱动显示器逐行刷新。

第五次作业：MIPS & 实时系统

MIPS系统的CPU运行状态定义了三种运行状态，用于实现操作系统保护和安全隔离。分别是核心态（kernel mode）、监管态（supervisor mode）、用户态（user mode），区别是能否执行特权指令。

状态转换通过异常/中断、系统调用、kernel返回user这几种方式在状态间切换。

对实时系统的理解：实时系统是指必须要在严格时间约束内对外部事件做出响应的计算系统，其正确性不仅取决于逻辑结果，还在于是否能在截止时间前完成响应。

分为硬实时（错过截止时间会造成严重后果），比如飞控、汽车刹车，与软实时（偶尔错过截止时间可以接受，只是性能下降），比如视频播放、游戏引擎等。

有以下关键特性：可预测性、确定性响应、优先级调度、低中断延时、资源隔离。

第六次作业：可重入

以C为例。

可重入是指可以在其执行过程中被中断，然后再次安全的进入该函数，而不会导致数据错误/未定义；

不可重入是指在重入时可能会因为共享状态而导致错误结果。

不可重入，说白了就是依赖非原子的、持久的内部状态：

```
char* itoa(int n)
{
    static char buf[16];
    sprintf(buf, "%d", n);
    return buf;
}
```

在多线程中调用两次：

```
char *a = itoa(123);
char *b = itoa(456);
printf("%s %s\n", a, b);
```

后一次会覆盖前一次。

将其改为可重入也很简单，如让外部调用者提供缓冲区：

```
char *itoa(int a, char* buf, size_t size)
{
    sprintf(buf, size, "%d", n);
    return buf;
}
```

也就是说，可重入函数要不使用静态/全局变量，不返回指向静态数据的指针、不调用不可重入的函数、所有数据都来自参数、局部变量或调用者提供的存储。

第七次作业：地址空间

变量/代码	所属段
x1	.bss
x2 = 3.14	.data
a1 = 10	.data
b1	.bss
str1[32] = "abcde"	.data
str2[32]	.bss
add()	Text
add内的a b ss temp	Stack
add内的static j = 1000	.data
main()	Text
main内的v1 v2 x strp str[32]	Stack
main内的staic int ss	.bss
malloc(100)分配内存	Heap

第八次作业：页式管理

"每个进程拥有自己独立的地址空间"实现：

在页式存储管理中，每个进程都被赋予独立的虚拟地址空间，这个机制主要依赖于页表（page table）和内存管理单元（mmu）来实现。

操作系统会为每一个进程分配一个连续的虚拟地址空间，通过页表来映射到物理地址。页表记录了哪些虚拟页面已经被分配。由于每个进程都有独立页表，即使两个进程使用相同的虚拟地址，它们会映射到不同的物理地址。

“构造两个进程的页表结构”

假设逻辑地址为32位，每页大小为4KB，使用低12位作为页内偏移量，剩余20位用于表示页号。采用二级页表，将这20位等分，前10位作为顶级页表索引，后10位表示二级页表索引。

因此可以假设两个进程各自的页表：
进程1：

顶级页表：

INDEX	FRAME
0x000	0x1A0
0x001	0x2B1

二级页表：

INDEX	FRAME
0x000	0x3C2
0x001	0x4D3

进程2：

顶级页表：

INDEX	FRAME
0x000	0x5E4
0x001	0x6F5

二级页表：

INDEX	FRAME
0x000	0x7G6
0x001	0x8H7

这样，即使两个进程都访问同一个逻辑地址，但是最后映射到物理地址上是不会重合的。