Jos中进程切换和线程切换：

进程切换分两步：

1.切换页目录以使用新的地址空间  
2.切换内核栈和硬件上下文。  
  
对于linux来说，线程和进程的最大区别就在于地址空间。对于线程切换，第1步是不需要做的，第2是进程和线程切换都要做的。所以明显是进程切换代价大

线程上下文切换和进程上下文切换一个最主要的区别是线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的。这两种上下文切换的处理都是通过操作系统内核来完成的。内核的这种切换过程伴随的最显著的性能损耗是将寄存器中的内容切换出。

另外一个隐藏的损耗是上下文的切换会扰乱处理器的缓存机制。简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。还有一个显著的区别是当你改变虚拟内存空间的时候，处理的页表缓冲（processor’s Translation Lookaside Buffer (TLB)）或者相当的神马东西会被全部刷新，这将导致内存的访问在一段时间内相当的低效。但是在线程的切换中，不会出现这个问题。

<https://blog.csdn.net/xiangwanpeng/article/details/78196539>

jos中进程的创建

而对于jos好像ENVCREAET()创建的比较像进程，而ENVCREATE()创建的函数再调用fork()函数创建的像线程。但是无论是谁创建的都是生成一个新的环境。而且在进行任务切换的时候都是使用env\_run()函数进行切换的，看起来与linux并没有什么不同。

而且在user/dumbfork.c运行的时候始终没有看懂UTEXT,UTEMP这些变量定义在哪？而且是如何实现了copy\_on\_write技术的。

Jos中进程切换具体步骤：

1. 设置好IDT表（放置中断和异常处理函数）
2. 系统中断发生，调用中断表中的函数  
   系统发生中断或者异常的时候，硬件会往寄存器中保存变量，特权和非特权级的转移保存的变量不同<https://pdos.csail.mit.edu/6.828/2018/readings/i386/s09_06.htm>

然后中断表中的函数将中断号和错误码（如果硬件没有压入的话，作为padding压入）压栈，之后中断表中断的函数call统一的中断处理函数\_alltrap

1. \_alltrap push ds push es push sp（都是程序当前的寄存器状态）

Call trap(c语言处理函数)

1. Trap(struct Trapframe\* tf)根据传入的参数进行统一的处理，根据中断号或者异常号来进行中断的分发处理

truct Trapframe {

struct PushRegs tf\_regs;

uint16\_t tf\_es;

uint16\_t tf\_padding1;

uint16\_t tf\_ds;

uint16\_t tf\_padding2;

uint32\_t tf\_trapno;

//当发生中断或者异常的时候，硬件选择性保存的变量

// below here defined by x86 hardware

uint32\_t tf\_err;

uintptr\_t tf\_eip;

uint16\_t tf\_cs;

uint16\_t tf\_padding3;

uint32\_t tf\_eflags;

// below here only when crossing rings, such as from user to kernel

uintptr\_t tf\_esp;

uint16\_t tf\_ss;

uint16\_t tf\_padding4;

} \_\_attribute\_\_((packed));