北京大学

PEKING UNIVERSITY

XV6 进程线程源码阅读报告

组长：	罗登 2001210364
组员 1：	毕廷竹 2001210186
组员 2：	周旭敏 2001210723
组员 3：	罗旭坤 2001210368
组员 4：	张志鑫 2001210547

XV6进程线程报告

基本问题回答

问题一

1.什么是进程，什么是线程？操作系统的资源分配单位和调度单位分别是什么？XV6中的进程和线程分别是什么？都实现了吗？

进程是计算机中的程序关于某数据集合上的一次运行活动，进程是程序本身和记录程序各种信息的数据结构的集合，进程是一个活动起来的程序。

线程的概念依赖于进程而存在，在操作系统的设计过程中，设计者发现，不仅进程之间有并发的要求，进程之间更小的模块也有并发的要求，所以诞生了线程的概念。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务，进程之内的线程是资源共享的。

在一个支持线程的操作系统中，操作系统资源分配的单位是进程，资源调度的最小单位是线程。

XV6中有进程的概念而没有线程的概念。

XV6中进程定义的结构体proc如下所示：

```
struct proc {
  uint sz;                     // Size of process memory (bytes)//进程内存的大小
  pde_t* pgdir;                // Page table//页表的指针
  char *kstack;                // Bottom of kernel stack for this process//内核
  enum procstate state;        // Process state //进程的状态 六种之一
  volatile int pid;            // Process ID //进程ID
  struct proc *parent;         // Parent process //父进程
  struct trapframe *tf;        // 当该进程启动系统调用的时候应该保存的信息
  struct context *context;     // 进程切换应该保存的寄存器信息
  void *chan;                  // If non-zero, sleeping on chan //阻塞位的标志
  int killed;                  // If non-zero, have been killed //被kill的标志位
  struct file *ofile[NOFILE];  // Open files //打开的文件表
  struct inode *cwd;           // Current directory //当前目录
  char name[16];               // Process name (debugging) //程序名字
};
```

在Unix系统中，其PCB结构也叫做proc，从此可以看出这两个系统之间的一些联系。在操作系统理论课程中，进程PCB是非常重要的一个数据结构，几乎包含了进程的所有信息。但是通过源码阅读可以发现，在xv6中的proc结构只有13行，尽管其中包含一些结构体指针，例如页表目录项pde_t，进程状态procstate，陷入帧trapframe，上下文context，打开文件表file*数组和索引节点inode等，其余的变量均比较简单，其中pid为进程唯一标识符，为int类型，其前面添加了一个volatile关键字，其字面意义是多变的。该关键字显式地告诉编译器变量在使用的时候从内存中取值，禁止了cpu cache，在一定程度上影响了性能，但是对于一致性要求高的内容，可以使用该关键字来修饰，这里确保了pid这个关键变量的正确性。从这个数据结构就可以看出xv6系统设计上的简单明了。

问题二

2.进程管理的数据结构是什么？在 Windows，Linux，XV6 中分别叫什么名字？其中包含哪些内容？操作系统是如何进行管理进程管理数据结构的？它们是如何初始化的？

进程管理的数据结构是进程控制块PCB，Process Controller Block。

PCB结构在不同的操作系统中有着不同的名字，所包含的内容也不一样，与系统设计者对功能、性能、安全的方面的考虑密切相关，但是也会包含一些必要的内容。

在 Windows 中定义进程的结构是 EPROCESS 和 KPROCESS 这两个结构。其中 EPROCESS 表示进程，KPROCESS 表示线程。按照微软的定义，Windows 中的进程EPROCEESS 简单地说就是一个内存中的可执行程序，提供程序运行的各种资源。进程拥有虚拟的地址空间，可执行代码，数据，对象句柄集，环境变量，基础优先级，以及最大最小工作集。
Windows 中的线程 KPROCESS 是操作系统处理机调度的基本单位。线程可以执行进程中的任意代码，包括正在被其他线程执行的代码。进程中的线程共享进程的虚拟地址空间和所申请的系统资源，每个线程拥有自己的例外处理过程，一个调度优先级以及线程上下文数据结构。线程上下文数据结构包含寄存器值，核心堆栈，用户堆栈和线程环境块。Windows 中的进程链表是一个双向的循环链表。这个环链表 LIST_ENTRY 结构把每个 EPROCESS链接起来。只要找到任何一个 EPROCESS 结构，就可以将整个链表遍历一遍，这就是枚举进程的基本原理。

在Linux系统中定义进程的结构是task_struct，主要包含了以下内容：

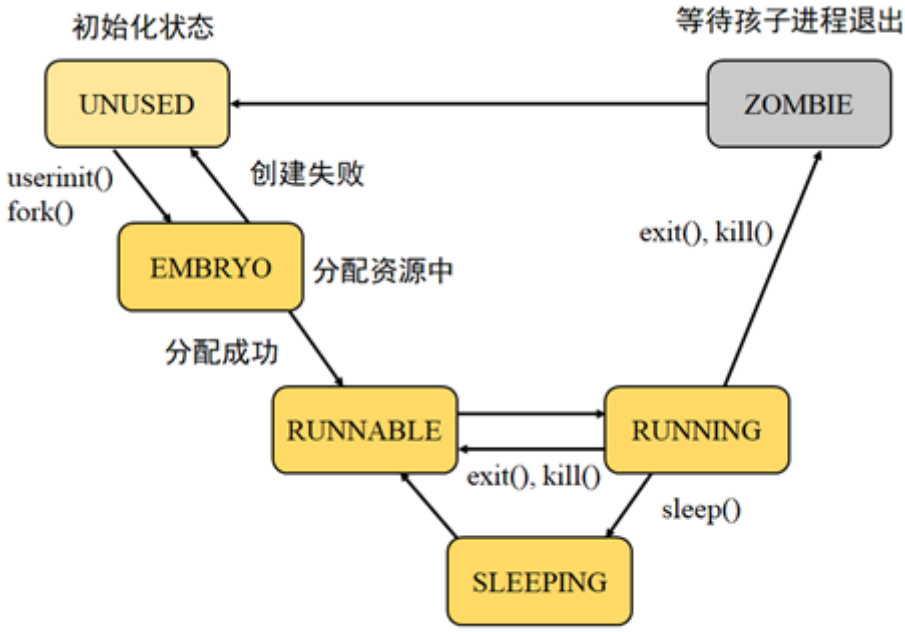
1. 标识符：标记进程信息的数字标识，包括pid（进程标识符）、uid（用户标识符）和pgrp（进程组标识）等。
2. 进程状态（state）：占有CPU执行、TASK_RUNNING、TASK_STOPPED、TASK_UNINTERRUPTBLE、TASK_INTERRUPTBLE和TASK_ZOMBIE。
3. 进程优先级：该进程的调度优先级，主要包括static_prio、normal_prio、prio和rt_priority四种。
4. 进程调度策略（policy）：主要包括SCHED_NORMAL（基于优先级的调度算法）、SCHED_FIFO（先进先出的调度算法）、SCHED_RR（时间片轮转调度算法）、SCHED_BATCH（用于非交互的CPU消耗型进程的调度算法）和SCHED_IDLE（用于系统负载低时的调度算法）五种。
5. 进程队列双向链表指针：包括next_tack和prev_task。
6. 进程关系指针：包括real_parent（父进程）、parent（接收终止信号的父进程）、children（子进程链表）、slibing（兄弟进程链表）和group_leader（所在进程组的领头进程）。
7. 内存信息：包括程序代码和进程相关数据的地址，如start_code（代码段地址）、end_code（代码长度）等。还包括和其它进程共享的内存块指针。
8. 上下文数据：进程执行时寄存器中保存的数据，主要保存在一个名为tss的结构中。
9. I/O状态信息：包括显示的I/O请求，分配给进程的I/O设备和打开文件表，如files指针指向打开文件链表。
10. 时间信息：进程运行的相关时间信息，如utime（用户态运行时间）、stime（系统态运行时间）和start_time（进程开始运行时刻）等。

XV6中为proc，类似Unix，在第一个问题中已经查看了proc的具体结构。

问题三

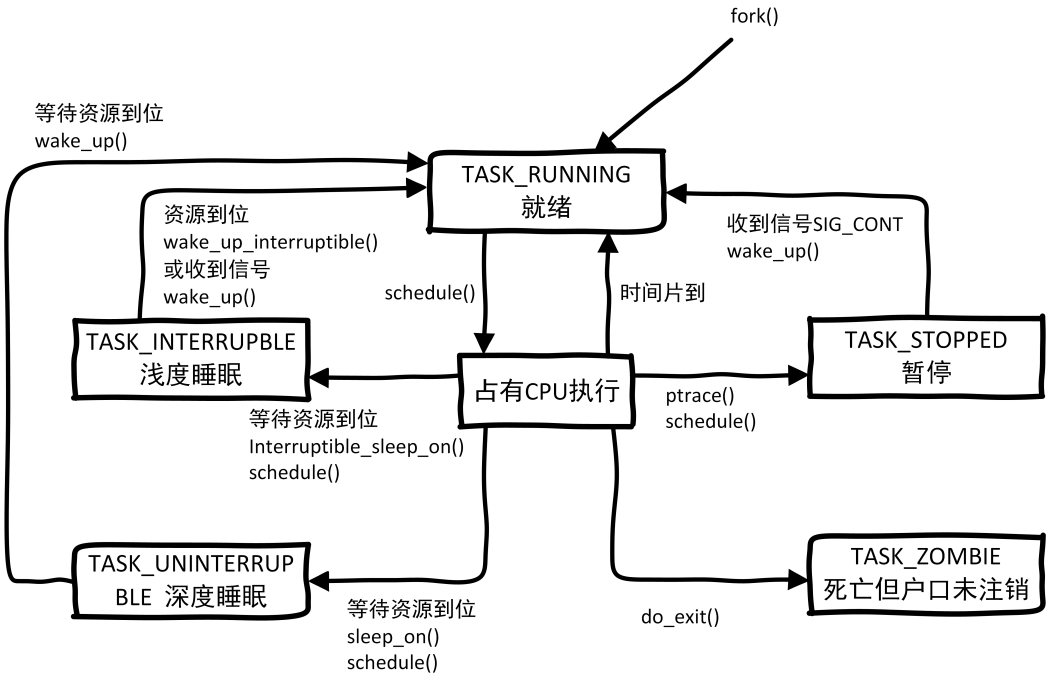
3.进程有哪些状态？请画出 XV6 的进程状态转化图。在 Linux，XV6 中，进程的状态分别包括哪些？你认为操作系统的设计者为什么会有这样的设计思路？

进程的最基本的就是三状态：运行态，就绪态，阻塞态。在XV6中，设计了6种进程的状态，分别是UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE。XV6中的进程状态转换如下图所示：



XV6进程状态转换图

在Linux中，也设计了6种进程的状态，分别是：TASK_RUNNING，TASK_INTERRUPTIBLE，TASK_UNINTERRUPTIBLE，TASK_STOPPED or TASK_TRACED，TASK_DEAD - EXIT_ZOMBIE，TASK_DEAD - EXIT_DEAD。Linux的进程状态转化图如下：



从最基本的三状态到六状态，操作系统更加精细的区分了各个进程处在的不同状态，可以更好地管理每一个进程，提高整个操作系统的资源利用率。

问题四

4.如何启动多进程（创建子进程）？如何调度多进程？调度算法有哪些？操作系统为何要限制一个 CPU 最大支持的进程数？XV6 中的最大进程数是多少？如何执行进程的切换？什么是进程上下文？多进程和多 CPU 有什么关系？

在XV6中，通过fork()函数就可以创建子进程了。

调度多进程主要有三个方面：调度时机，调度操作，调度算法。调度时机有很多，例如：时钟中断，进程意外退出，进程等待IO操作等等。实际上，调度时机可以用一句话概括：内核对中断/异常/系统调用处理后返回到用户态前。调度操作指的就是对旧的进程和新的进程的操作，包括保存旧进程的上下文环境，修改旧进程PCB的状态，切换内核栈，切换全局页目录等等。调度操作概括为：保存旧的信息，设置新的信息。调度算法有很多，例如：先来先服务调度算法FCFS，短作业（进程）优先调度算法SJF（非抢占）/SPF（抢占）高优先权优先调度算法HPF，时间片轮转算法，多级反馈队列算法FB。这些调度算法，体现的是一种思想。操作系统的设计者希望CPU能被高效的利用，也让每个进程相对公平的得到上CPU的机会。

进程是资源分配的单位，操作系统中的资源是有限的，所以必须要限制系统中进程的数量。一旦进程数量过多且都不释放资源，计算机就会死机。

XV6中最大进程数量是64个。

进程的切换就是保存旧的信息，设置新的信息。进程的上下文是进程运行依赖的各种环境，例如各个寄存器的值。

多进程和多CPU的关系：一个进程同一时间只能在一个CPU上运行，一个进程在不同的时间可以处在不同的CPU上。CPU的数量一般总是远小于进程的个数的，所以调度算法依然需要相对公平的去解决多进程上多CPU的问题。

多处理器调度的特征：多个处理器共同组成一个多处理机系统；处理器之间可以负载均衡。对称多处理器（SMP）调度，每个处理器运行自己的调度程序，调度程序对共享资源的访问需要同步。

多处理器调度不仅要决定选择哪一个进程执行，还要决定在哪一个CPU上执行，需要考虑进程在多个CPU之间迁移的开销，要尽可能的让进程总在同一个CPU上面执行，要考虑到负载均衡问题。

问题五

5.内核态进程是什么？用户态进程是什么？它们有什么区别？

以32位的X86的Linux操作系统来举例，Linux中虚拟地址空间的大小为4GB。从高地址向低地址看虚拟地址空间，前面1GB空间是内核空间，后面的3GB空间是用户空间。进程运行在内核空间的时候称为内核态，运行在用户空间的时候称为用户态。对于用户态进程来说，它可以通过系统调用或者中断的方式进入内核态。而有一部分进程是永远处于内核状态的，它们不会跳转到用户空间执行，一直处在内核区域执行。内核进程，没有独立的地址空间，所有内核线程的地址空间都是一样的，没有自己的地址空间，其运行在内核空间，本身就是内核的一部分或者说是内核的分身。用户态进程和内核态进程最大的区别就在于权限不同，用户态进程的权限级别为3，内核态进程的权限级别为0。用户态进程陷入内核态的时候，需要发生栈的转换，也就是从用户栈转换为内核栈。用户态进程要在TSS寄存器中找到内核栈的栈指针和栈底，通过修改ESP和EBP成功的切换成内核栈。而从内核栈切换为用户栈，用户栈的栈指针和栈底是记录在内核栈里面的，所以不需要借助TSS寄存器。

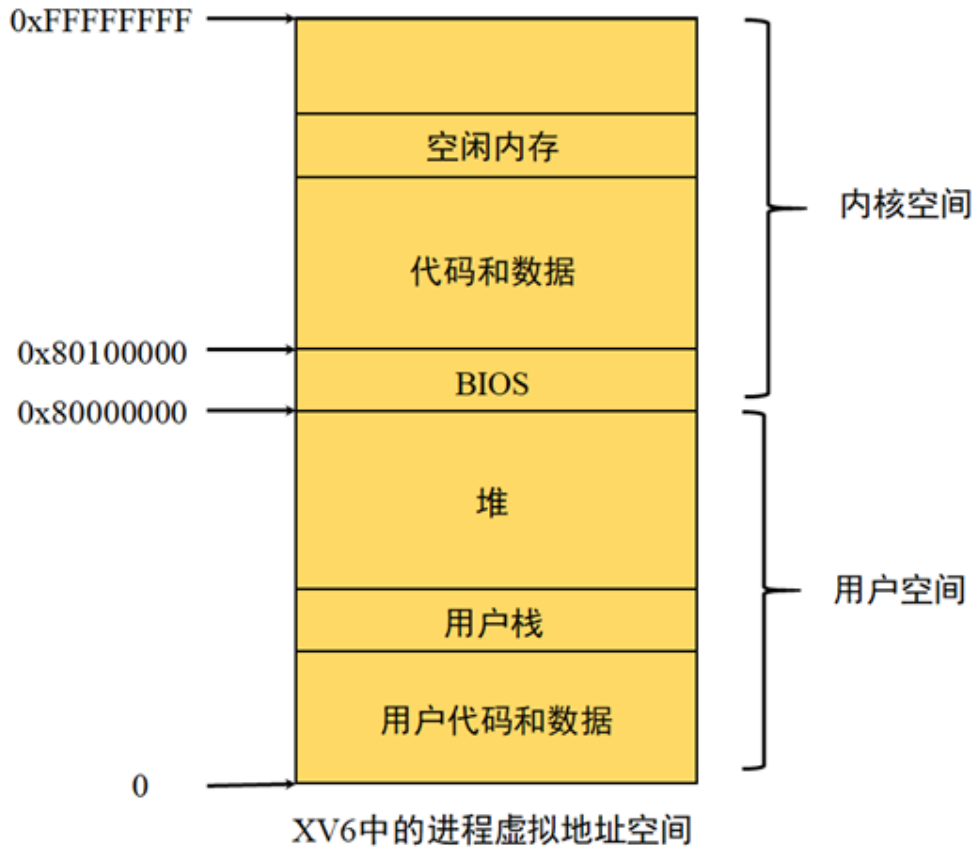
Linux所谓的用户态和内核态，本质是对CPU提供的功能的一层封装抽象。现代CPU，其设计目标主要是为了完美高效的实现一个多任务系统，多任务系统的三个核心特征是：权限分级、数据隔离和任务切换。以X86_64架构为例，权限分级通过CPU的多模式机制和分段机制实现，数据隔离通过分页机制实现，任务切换通过中断机制和任务机制（TR/TSS）实现。内核态和用户态的概念，是Linux为了有效实现CPU的权限分级和数据隔离的目标而出现的，是通过组合CPU的分段机制+分页机制而形成的。还是以X86_64架构为例，在当CPU处于保护模式下时（X86_64CPU有5种模式，保护模式是其中之一，此时CPU.CR0.PE=1），当CPU.CS=系统代码段时（CS.CPL=0）为内核态，此时通过CPU的指令有操控全部寄存器的权限（包括FLAGS和CR寄存器），当CPU.CS=用户代码段时（CS.CPL=3）为用户态，此时通过CPU的指令只有操控部分寄存器的权限。

问题六

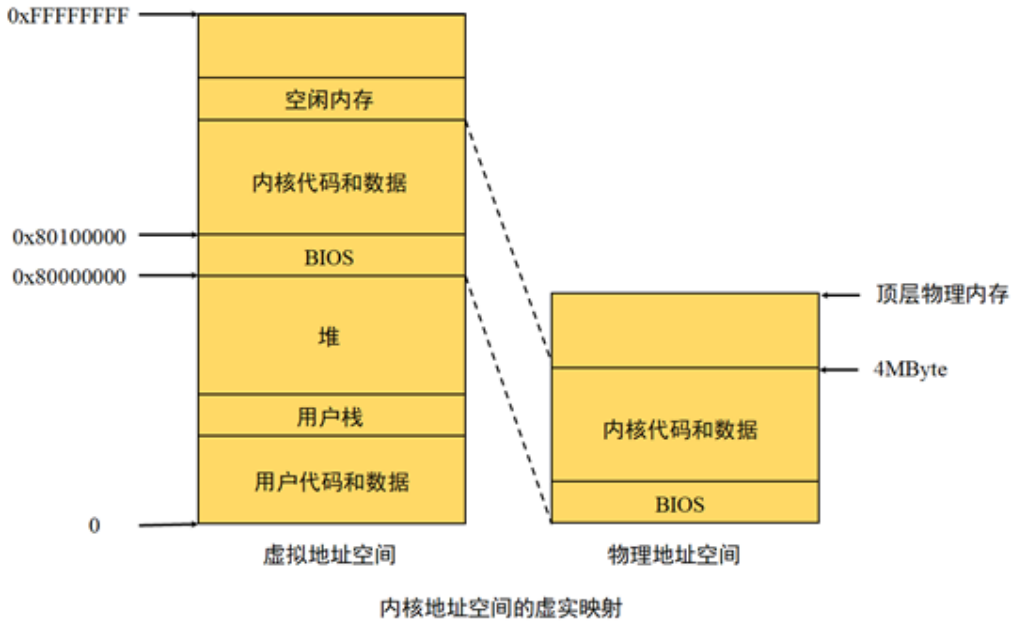
***6.**进程在内存中是如何布局的，进程的堆和栈有什么区别？*

在x86架构的32位机器，Linux操作系统中，进程空间由下到上依次是：保留区，只读区，读写区，堆区，动态链接库区，栈区，内核区。进程的堆存放的是malloc分配的变量，进程的堆区存放的是函数调用时要传递的参数和临时变量。所以以前写C++算法的时候：要把声明的大数组放在函数外面成为全局变量，而不是放在函数里面，这样的话数据就放在了数据区域而不是比较小的栈区域了

下面有两张图展示了进程在内存中的布局：



XV6中的进程虚拟地址空间



内核地址空间的虚实映射

代码细节和分析

types.h：记录了一些变量的别名。

param.h：设置了XV6操作系统中的参数。

memlayout.h：内存布局的情况。其中涉及到未开启分页状态下虚拟地址和物理地址之间的转换：

```
//下面的都是虚拟地址和物理地址之间的转换
//v2p 虚拟地址转换为物理地址
//p2v 物理地址转换为虚拟地址
//为什么要这么转换？
//启动分页了，但是页表还没有设置好，所以这个时候虚拟地址和物理地址之间的转换是线性的转换……
#ifndef __ASSEMBLER__

static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }

#endif

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as V2P, but without casts

```

defs.h：定义了各种声明和全局函数，具体的实现在各个函数板块中都有实现。

X86.h：将常用的汇编指令进行了C封装。

asm.h: 汇编程序宏来创建x86段，地址边界一定是4K的整数倍。

mmu.h: 定义了一些寄存器的值，一些数据结构，x86 memory management。

elf.h: 定义了ELF可执行目标文件的格式，魔数，程序头表，ELF头表。

proc.h：进程头文件，定义了XV6关于进程的数据结构，调度函数，多CPU多进程等信息。例如里面记录了XV6进程的6个状态如下：

```
//进程的六种状态
//未使用，胚胎(初期)，睡眠，可运行，正在运行，僵尸
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

proc.c: 进程控制的C程序文件，利用了proc.h中的数据结构完成相应的进程操作。从proc.c中的代码可以看出，XV6中组织进程的方式非常简单，就是一个长度为64的结构体数组。

```
struct {
  struct spinlock lock;//互斥锁
  struct proc proc[NPROC];//NPROC=64
} ptable;//进程表
//进程索引表，64个进程以数组的形式记录
```

Swtch.s: 上下文切换的汇编代码。

kalloc.c：物理内存的申请和释放。

小组讨论问题和回答

讨论一

1.Windows中的Eprocess中的E代表什么？

E代表Execute。

讨论二

2.僵尸进程和孤儿进程的区别？

任何一个子进程(init除外)在exit()之后，并非马上就消失掉，而是留下一个称为僵尸进程(Zombie)的数据结构，等待父进程处理。僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

讨论三

3.为什么要有僵尸状态？详细的解释一下僵尸进程？

给进程设置僵尸状态的目的是维护子进程的信息，以便父进程在以后某个时间获取。这些信息包括子进程的进程ID、终止状态以及资源利用信息(CPU时间，内存使用量等等)。如果一个进程终止，而该进程有子进程处于僵尸状态，那么它的所有僵尸子进程的父进程ID将被重置为1（init进程）。继承这些子进程的init进程将清理它们(init进程将wait它们，从而去

除僵尸状态)。

由于子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程 到底什么时候结束. 那么会不会因为父进程太忙来不及wait子进程，或者说不知道 子进程什么时候结束，而丢失子进程结束时的状态信息呢? 不会。因为UNIX提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息， 就可以得到。这种机制就是： 在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。但是仍然为其保留一定的信息（包括进程号the process ID，退出状态the termination status of the process，运行时间the amount of CPU time taken by the process等）。直到父进程通过wait / waitpid来取时才释放. 但这样就导致了问题，如果进程不调用wait / waitpid的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程. 此即为僵尸进程的危害，应当避免。

讨论四

4.内核态和用户态如何标记？区别是指令级别的吗？

内核态和用户态的区别是用硬件来区别的，限制操作系统在用户态下的访问也是由硬件电路完成的。

讨论五

5. 一个block多大？disk block 和页对应吗？页和块的概念？

页和块的对象不同：页是对逻辑地址进行分页存储，块是对实际地址进行分块存储。页号是虚拟地址的划分，指向程序中的某一页，每个页号对应一个页面号。块号是实际地址的划分，指向内存空间中某一个物理块。

讨论六

6. v2p为什么可以直接减常数？

启动操作系统时，内核段系统内存可以直接定位。

讨论七

7. lgdt/lidt是什么？

这两条都是汇编层次的指令：加载全局/中断描述符表格寄存器。对应的寄存器是GDTR和IDTR。

讨论八

8.ltr/tss是什么？

LTR指令是专门用于装载任务状态段寄存器TR的指令。

在任务内发生特权级变换时堆栈也随着自动切换，外层堆栈指针保存在内层堆栈中，而内层堆栈指针存放在当前任务的TSS中。所以，在从外层向内层变换时，要访问TSS(从内层向外层转移时不需要访问TSS，而只需内层栈中保存的栈指针)。

讨论九

9. CR0-CR3是什么？

在x86_32位机器上有四个32位的控制寄存器，它们是CR0，CR1，CR2和CR3。

CR0中包含了6个预定义标志，0位是保护允许位PE(Protedted Enable)，用于启动保护模式，如果PE位置1，则保护模式启动，如果PE=0，则在实模式下运行。1位是监控协处理位MP(Moniter coprocessor)，它与第3位一起决定：当TS=1时操作码WAIT是否产生一个“协处理器不能使用”的出错信号。第3位是任务转换位(Task Switch)，当一个任务转换完成之后，自动将它置1。随着TS=1，就不能使用协处理器。CR0的第2位是模拟协处理器位 EM (Emulate coprocessor)，如果EM=1，则不能使用协处理器，如果EM=0，则允许使用协处理器。第4位是微处理器的扩展类型位ET(Processor Extension Type)，其内保存着处理器扩展类型的信息，如果ET=0，则标识系统使用的是287协处理器，如果 ET=1，则表示系统使用的是387浮点协处理器。CR0的第31位是分页允许位(Paging Enable)，它表示芯片上的分页部件是否允许工作。

CR1是未定义的控制寄存器，供将来的处理器使用。

CR2是页故障线性地址寄存器，保存最后一次出现页故障的全32位线性地址。

CR3是页目录基址寄存器，保存页目录表的物理地址，页目录表总是放在以4K字节为单位的存储器边界上，因此，它的地址的低12位总为0，不起作用，即使写上内容，也不会被理会。

讨论十

10. proc.c中的wakeup和wakeup1的区别是什么？

区别在于有没有加锁，wakeup在wakeup1的基础上加了锁。

讨论十一

11. swtch.S中汇编代码的详细解释

从C语言转入汇编语言进行操作，一定要设置好栈的位置，因为函数的参数保存在栈里面。4+%esp是第一个参数的位置，也就是·`**old`，是一个上下文指针的指针；第二个参数的位置在8+%esp，是上下文指针`*new`。

```
.globl swtch
swtch:
  movl 4(%esp), %eax//利用栈来传递传参数，同样是32位的机器才会这样传递参数
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)//一个打了括号 一个不打括号 修改指针本身的内容 间接寻址//void swtch(struct context **old, struct context *new);
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

讨论十二

12. 实模式和保护模式下的寻址方式？

以x86_32位机器来举例，刚刚开机的时候处于保护模式，地址生成是CS<<4+IP，生成的地址是一个20位的地址，寻址空间就是1M。机器正式启动之后，CS就不能用左移四位加上IP的方式生成地址了，而要在保护模式下进行地址生成。CS段寄存器要去找到段描述符表，段描述符表中的每一项如下所示。CS段寄存器找到了对应的表项之后，根据表项中的内容生成32位的地址（base_15_0，base_23_16，base_31_24）。在表项中生成的32位地址加上32位的EIP就真正生成了一个32位的地址。

```
// Segment Descriptor 段描述符
//看到这个数据结构，我终于对32位机器上的地址生成有了直观的感受……
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};

```

讨论十三

13. 内核空间和内核栈被所有进程共享吗？为什么？

内核空间

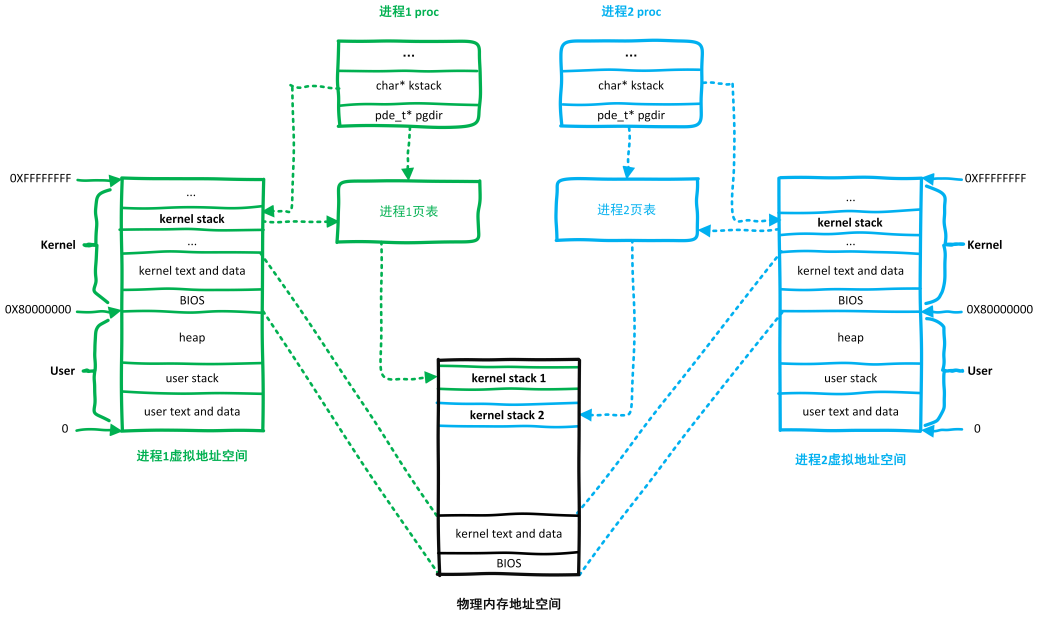
内核空间表示运行在处理器最高级别下的代码或数据，在XV6中占据0X80000000到0XFFFFFFFF的进程虚拟存储空间。如下图所示，内核空间中的BIOS和kernel code and data由所有进程共享，但是不对称的共享，只有运行在内核态的进程才可以通过系统调用切换到内核态访问内核空间，进程运行在内核态时产生的地址都属于内核空间。

内核空间共享，是为了给用户进程提供系统调用的接口，用户执行系统调用，内核根据系统调用的编号跳转到对应的内核地址进行执行，然后返回用户模式。这样用户级进程中的应用程序才能通过共享的内核空间来和内核交互，以及调用其他功能和访问硬件。

内核栈

每个进程拥有各自的内核栈空间，并不共享。在XV6中的PCB结构（proc）中，kstack指向了该进程内核栈的栈基址，并且通过kalloc()方法为进程在内核虚拟地址空间中分配一个4KB大小的空间作为内核栈。

进程的内核栈与用户栈的作用类似。每个进程运行时都持有上下文，为了保证内核态和用户态上下文的隔离，在陷入内核时不影响用户态，所以使用了不同的栈。当进程在用户地址空间中执行的时候，使用的是用户栈，CPU堆栈指针寄存器中存的是用户栈的地址；当进程在内核空间执行时，使用的是内核栈，CPU堆栈指针寄存器中放的是内核栈的地址。因此，用户态使用的用户栈和内核态使用的内核栈并无本质区别，它们均处于同一块页表映射中，但内核栈处于高特权级访问限制的虚拟地址中，防止用户态代码访问内核数据。



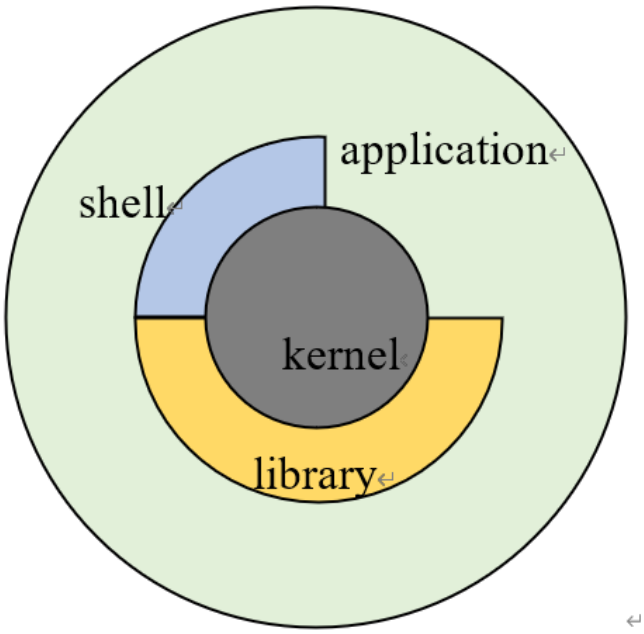
讨论十四

14.操作系统是如何区分用户态和内核态的？是硬件层面（指令级别）还是软件层面？

这个区分是硬件层面的，是通过CPU寄存器上的标志位来区分的。最开始提出这个问题是因为联想到了计算机组成原理中的指令译码、执行等相关知识，其中有关于权限等级的标识。因此在小组讨论的时候提出，在这里给出一个更书面化的回答。

用户态和内核态是进程的两种运行级别，一个运行在内核态的进程可以执行指令集中的任何指令，并且可以访问系统中的任何存储位置。用户态的进程不允许执行特权指令，比如停止处理器、改变模式位，获取进行I/O操作，也不允许用户模式中的进程直接引用内核地址空间的代码和数据。

在Intel X86 CPU中，用Ring0到Ring3四种不同的执行等级，以Linux系统为例，只使用了其0级和第3级表示内核态和用户态。



那么CPU是如何区分内核态和用户态的呢？CS寄存器的最低两位表明了当前代码的特权等级，它们分别是cs和eip这两个寄存器。其中cs是代码段选择寄存器，eip是偏移量寄存器。

下面简述一下从用户态转到内核态的流程：

1. 用户态程序首先将一些数据值放在寄存器中, 或者使用参数创建一个堆栈(stack frame), 以此表明需要操作系统提供的服务。
2. 用户态程序执行陷阱指令（trap），陷入指令是实现系统调用的方式，只能在用户态执行。
3. CPU切换到内核态, 并跳到位于内存指定位置的指令, 这些指令是操作系统的一部分, 他们具有内存保护, 不可被用户态程序访问。
4. 这些指令称之为陷阱(trap)或者系统调用处理器(system call handler). 他们会读取程序放入内存的数据参数, 并执行程序请求的服务。
5. 系统调用完成后, 操作系统会重置CPU为用户态并返回系统调用的结果。

参考文献

[1] 布赖恩特, 奥哈洛伦, 奕利, 等. 深入理解计算机系统[M]. 中国电力出版社, 2004.

[2] Tanenbaum. 现代操作系统: 英文版[M]. 机械工业出版社, 2005.

[3] Stevens W R. UNIX 环境高级编程: 英文版[M]. 机械工业出版社, 2002.

[4] 哈工大李治军操作系统视频[EB/OL].https://www.bilibili.com/video/BV1d4411v7u7?from=search&seid=13140702690686790478, 2019-05-04

[5] 浅析task_struct结构体[EB/OL].https://blog.csdn.net/qq_41209741/article/details/82870876, 2018-09-27

[6] 怎样去理解Linux用户态和内核态[EB/OL].https://zhuanlan.zhihu.com/p/69554144, 2019-06-17

[7] 哈工大李治军操作系统视频[EB/OL].https://www.bilibili.com/video/BV1d4411v7u7?from=search&seid=13140702690686790478, 2019-05-04

[8] Windows 进程与线程管理[EB/OL].https://zhuanlan.zhihu.com/p/91616118, 2011-10-14

[9] 进程地址空间分布[EB/OL].https://blog.csdn.net/wangxiaolong_china/article/details/6844325, 2018-09-14

[10] 几个常用的操作系统进程调度算法[EB/OL].https://blog.csdn.net/qq_36136497/article/details/82697401, 2019-05-04

[11] 五种进程调度算法的实现[EB/OL].https://www.cnblogs.com/bajdcc/p/4707544.html, 2015-08-16