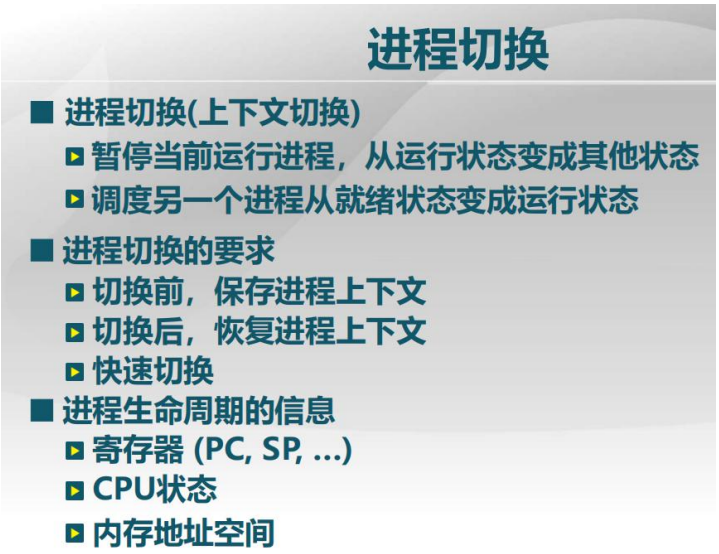
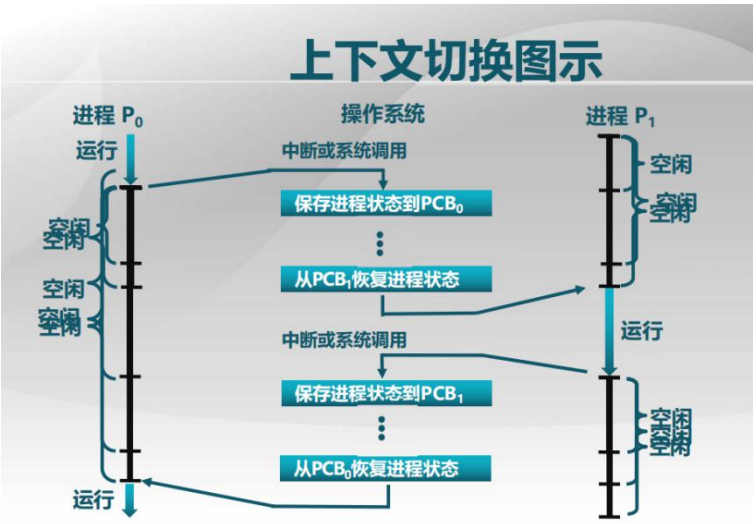
第一节 进程切换

第 12 讲讲涉及：①内核中的进程切换，即一个进程在运行过程当中，内核如何实现从一个进程到另一个进程的切换 ②进程创建③进程加载④进程等待与退出，是为用户提供的系统调用服务。



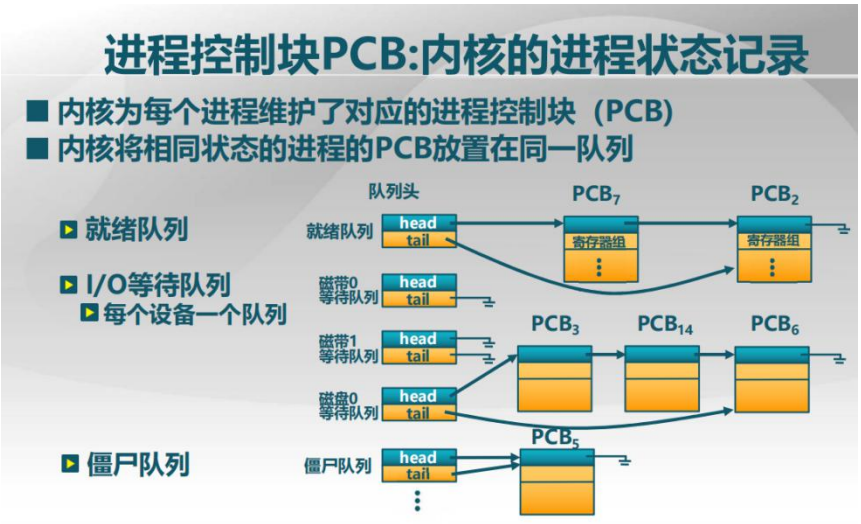
进程切换也称上下文切换，指的是暂停当前运行进程，将其从运行状态变成其他状态，这里其他状态可能会是由于进行 I/O 操作或者等待事件而进入等待状态，也可能是由于被抢先或者时间片用完转回到就绪状态。另一方面，是把当前进程停下来后，会调度另一个进程从就绪状态变成运行状态。进程上下文保存在 CPU 寄存器中。由于计算机系统中进程切换非常频繁，通常 10 毫秒左右就会有一次进程切换，这样，为保证系统运行效率，切换速度必须要非常快，因此通常由汇编实现。

具体说来，要保存的进程生命周期的信息如上图所示。通常情况下，进程切换时，内存地址空间里的这些信息是不会被另外一个进程所替代的，因此，内存地址空间中的内容大部分不用保存。



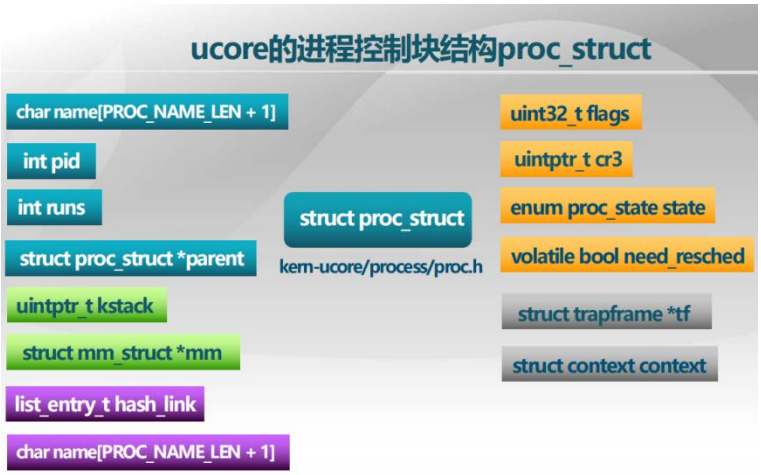
该图演示见 PPT，对其解释如下：假定系统中有 P0 和 P1 两个进程，首先，P0 处于用户态执行，执行过程当中，P1 处于空闲状态，可能是就绪，也可能是等待。执行过程中，P0 碰到一个系统调用或者中断，此处假定是时钟中断，则 P0 切换到内核，切换过程中，需保护现场到 PCB0(进程控制块 0)，把其当前执行状态都保存下来，此时需选择下一个就绪进

程，假定此时选定的下一个进程是 P1，则此时须在 PCB1 中恢复 P1 的现场，恢复完之后，进程便切换到进程 P1 了，此时 P1 开始运行，运行一段时间后，假定 P1 的时间片用完，又产生时钟中断，切回内核状态，此时保存进程 P1 的现场到其对应的进程控制块 PCB1，然后此时选定下一个进程，假设为 P0，这时恢复 P0 的现场，运行 P0.



PCB 记录的信息如上图所示。运行到退出状态(收尾状态)的进程进入僵尸队列。

到底有什么信息保存在进程控制块中？不同系统中进程控制块是不一样的，比如在 ucore 中：



Proc_struct 数据结构中保存了进程的相关信息。大致信息可分为下面几类：

①进程的标识信息：比如执行的是哪个可执行文件，进程的 id，父进程是谁等(蓝色)

②进程的执行状态信息：CPU 中状态寄存器的相关信息，地址空间的开头位置，一级页表的起始地址，进程的状态及其是否允许调度(橙色)

③进程所占用的资源：如占用的存储资源，所有分配给他的存储组织成相关的数据结构 mm，kstack 是占用的内核堆栈(绿色)

④保护现场用的：在中断及线程切换时，都需要保护现场，这些要保存到进程控制块中(灰色)

⑤进程在不同时刻处于不同状态，这些状态组成不同的队列，这里有几个指针结构用于描述当前进程到底在哪一个队列中(紫色)

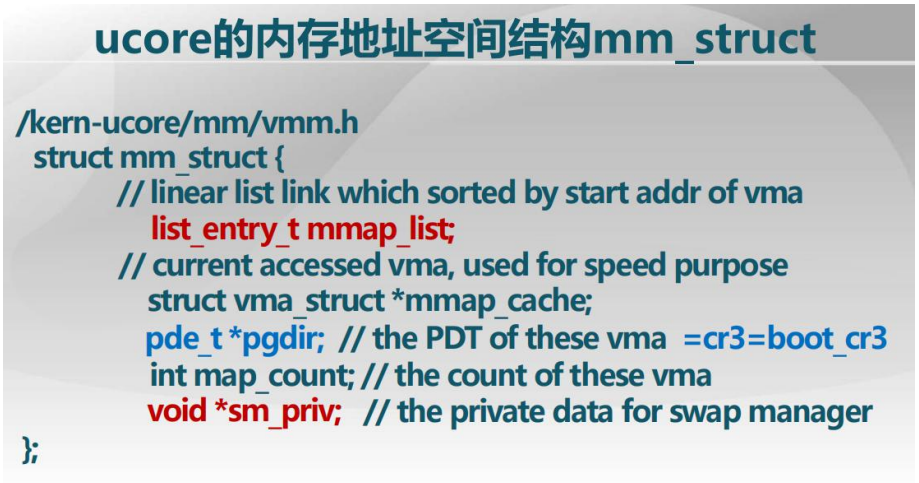
有了以上信息，我们对进程的执行状态便有了一个准确把握。

现在我们看 ucore 中进程控制块的数据结构：

在 ucore_lab\labcodes_answer\lab4_result\kern\process 的 proc.h 文件中有定义：

```
struct proc_struct {
    enum proc_state state;                      // Process state  进程状态
    int pid;                                    // Process ID  进程 id
    int tid;                                    //线程 id
    int gid;                                    //组 id
    int runs;                                   // the running times of Proces
    uintptr_t kstack;                           // Process kernel stack
    volatile bool need_resched;                 // bool value: need to be rescheduled
  to release CPU?  是否需要调度
    struct proc_struct *parent;                 // the parent process  父进程
    struct mm_struct *mm;                       // Process's memory management
field  进程内存管理数据结构
    struct context context;                     // Switch here to run process  进程现
场保护的上下文现场
    struct trapframe *tf;                       // Trap frame for current interrupt  进
程现场保护的中断保护现场
    uintptr_t cr3;                              // CR3 register: the base addr of
Page Directroy Table(PDT)  页表的起始地址
    uint32_t flags;                             // Process flag  标志位
    char name[PROC_NAME_LEN + 1];               // Process name  可执行文件的
进程的名字
    list_entry_t list_link;                     // Process link list   进程的链表
    list_entry_t hash_link;                     // Process hash list  进程的哈希表
};
```

这里需要特别说明内存地址空间的数据结构 mm_struct：



mmap_list 是内存地址空间链表。

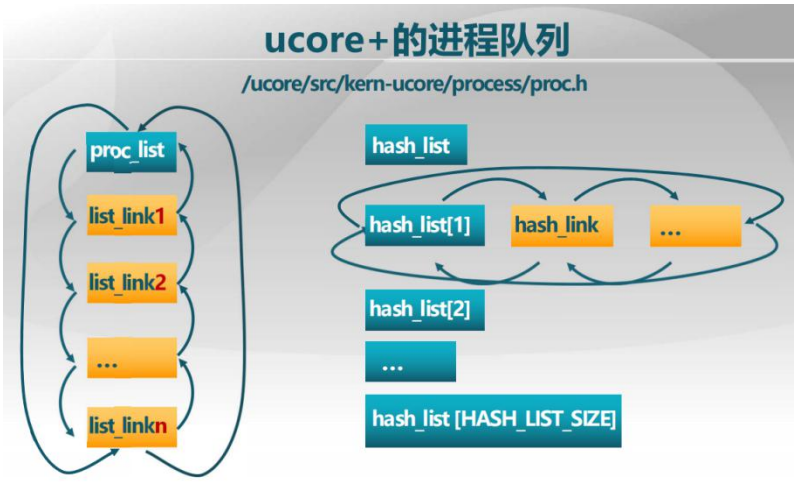
*pgdir 是第一级页表起始地址，即地址空间的起头。

map_count 是若有共享，共享了几次

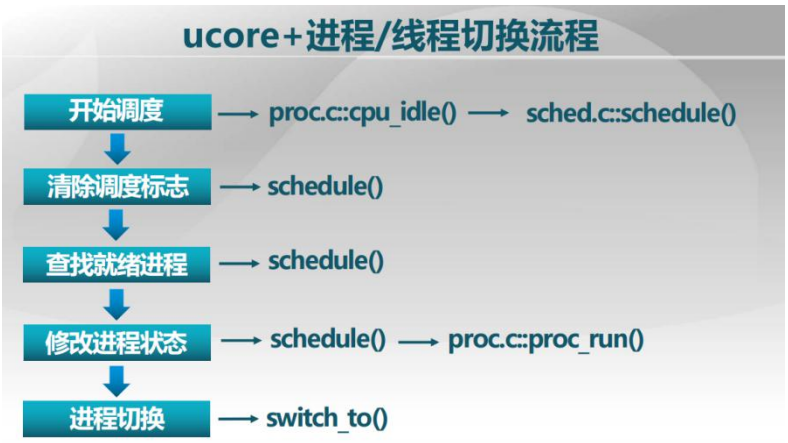
*sm_priv 是若与外存之间有置换，这个是关于置换的数据结构

代码在 vmm.h 中：

```
struct mm_struct {
    list_entry_t mmap_list;          // linear list link which sorted by start addr of vma  映射的链表
    struct vma_struct *mmap_cache; // current accessed vma, used for speed purpose
    pde_t *pgdir;                    // the PDT of these vma  页表的起始地址的指针
    int map_count;                   // the count of these vma  引用次数
    void *sm_priv;                   // the private data for swap manager
    int mm_count;                    // the number ofprocess which shared the mm
    lock_t mm_lock;                  // mutex for using dup_mmap fun to duplicat the mm
};
```



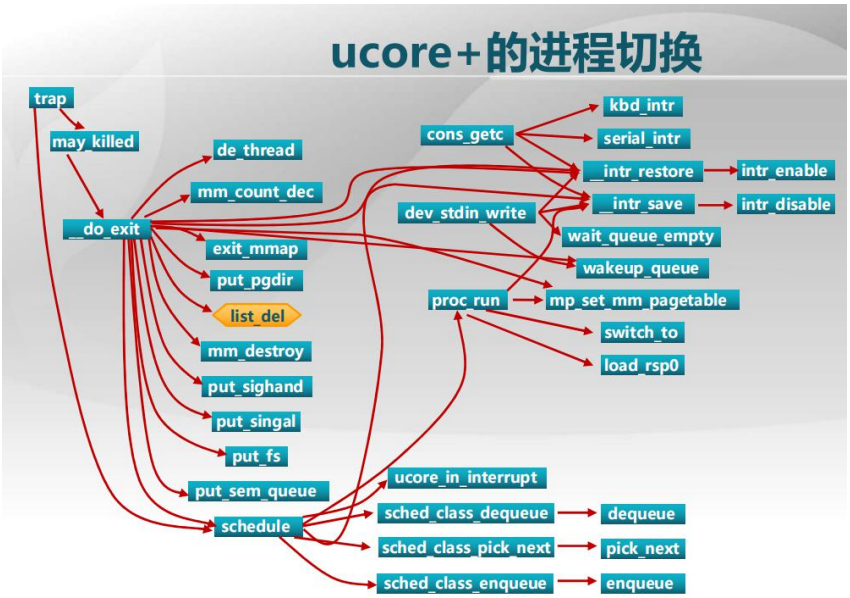
ucore 中的进程队列组织方法如上图，左面是双向链表，若链表很长，检索开销是很大的，所以 ucore 中又加了 hash list，即先加一级 hash 队列，然后每一级队列中，hash 值相同的再组成相应的自己的队列(右侧)。



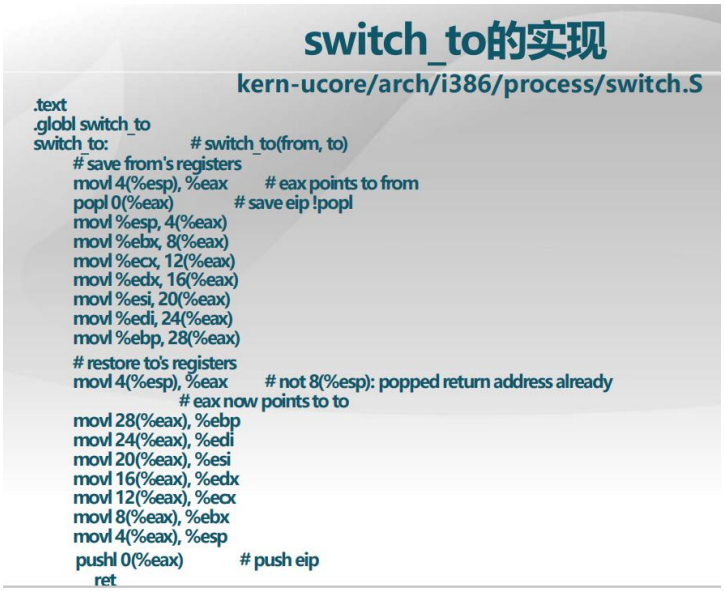
清除调度标志：表明现在正在调度，不能改调度标志了。

查找就绪进程：可能查找来的还是我自己。

修改进程状态：把上一个进程的状态改为就绪或等待，新进程改为运行。

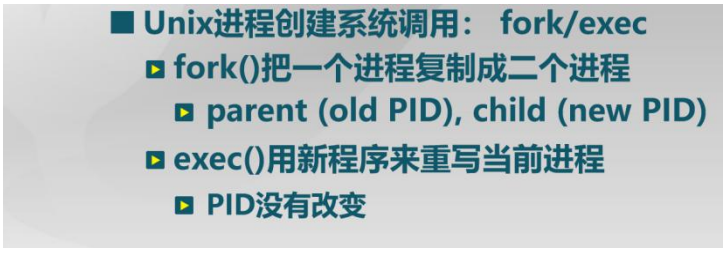


Switch_to 的代码是和平台相关的，每一个 CPU 平台上，所需保存的寄存器是不同的，为了保存/恢复的速度较快，所有的切换代码 switch_to 都是用汇编写的，大致就是前半段保护现场切换过去，即改 CR3(保存的现场信息)，下半段恢复现场，就继续执行了。



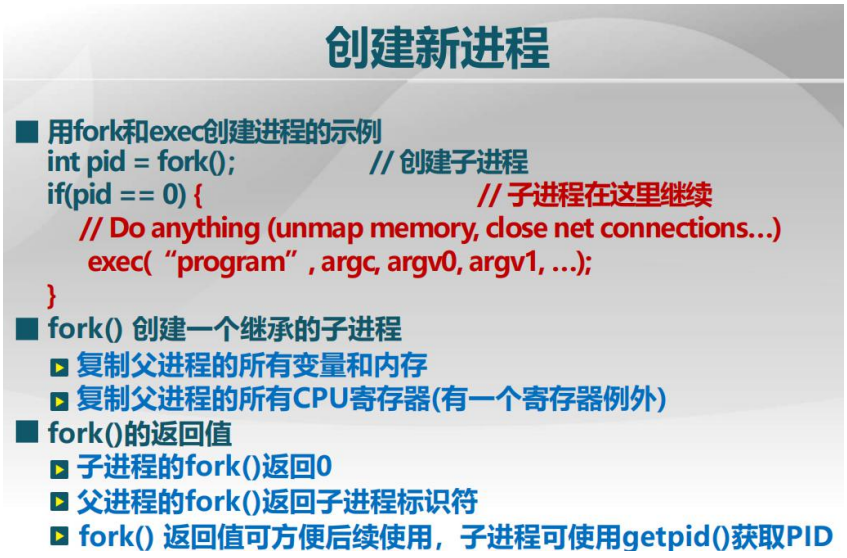
第二节 进程创建

进程创建是操作系统提供给用户使用的一个系统调用，完成新进程的创建工作，在不同的操作系统当中，进程创建的 API(或者说系统调用接口)是不一样的，如下图所示：

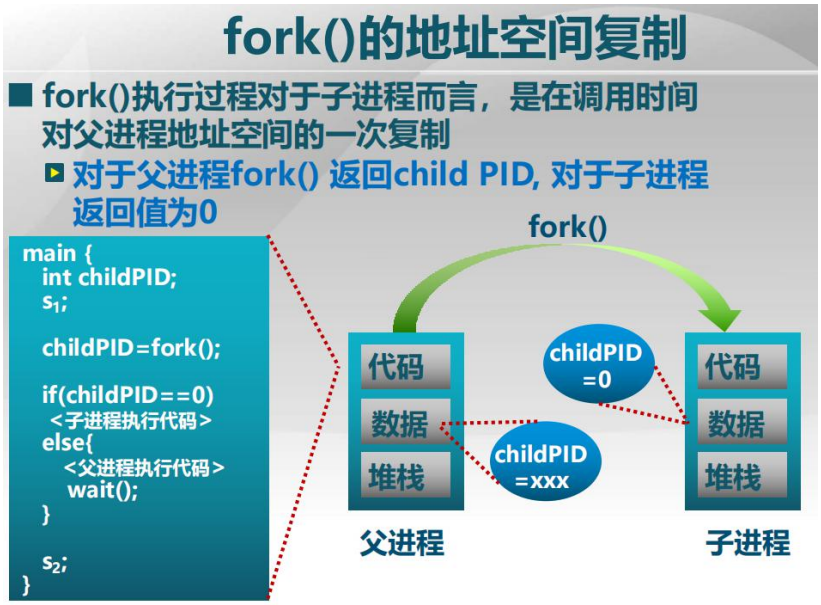


fork 完成把一个进程复制成两个进程，这时它们所执行的程序是一样的，但进程的 id 即 pid 不同，父进程中的是原来执行的那个进程的 id，子进程是分配给他的一个新 id。

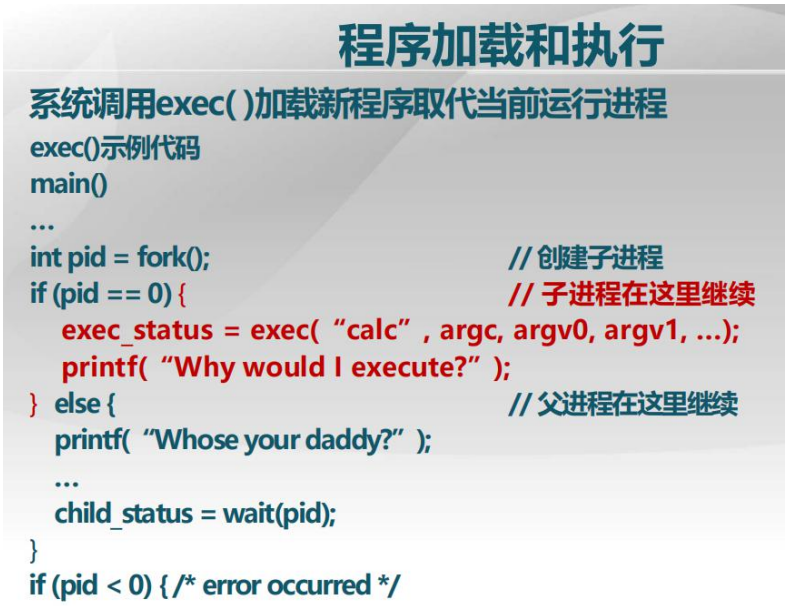
下图是一个实例：



fork()后，父进程和子进程都在 if 处开始执行(因为指令指针两者是相同的)，有一个寄存器例外，即存有 pid 的寄存器。

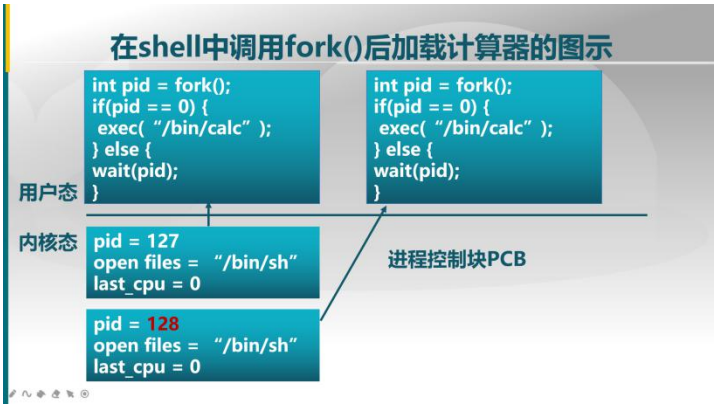


下面看程序的加载和执行：



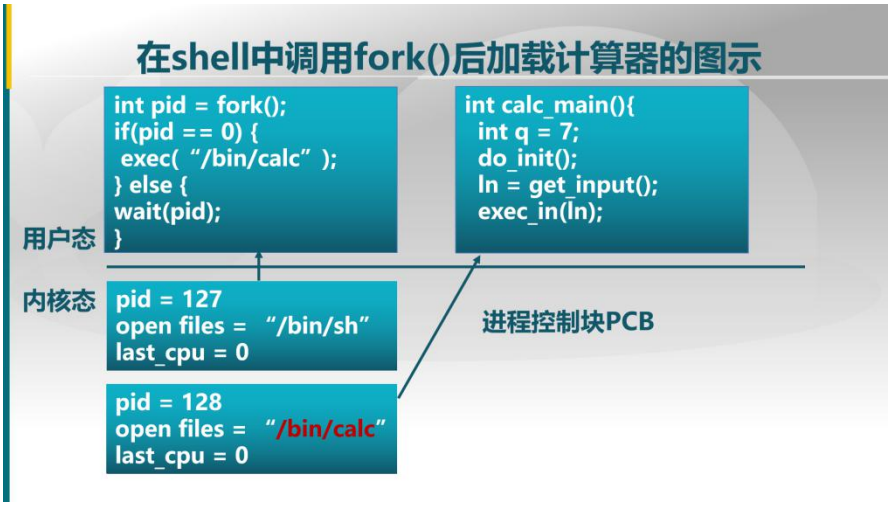
子进程利用 exec 指定可执行的程序及参数。

执行 fork 后操作系统中的变化如下：

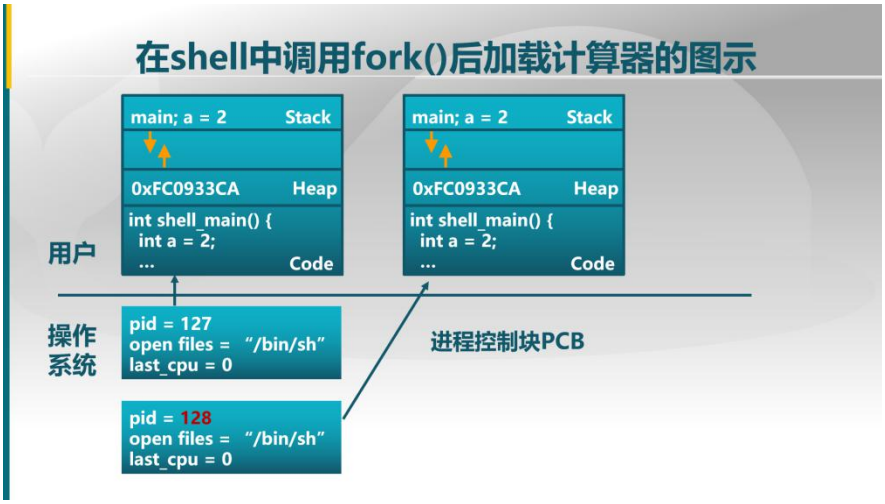


fork 执行时做了一个复制，复制后子进程的(实际)pid=128,然后父进程因为(变量)pid!=0，

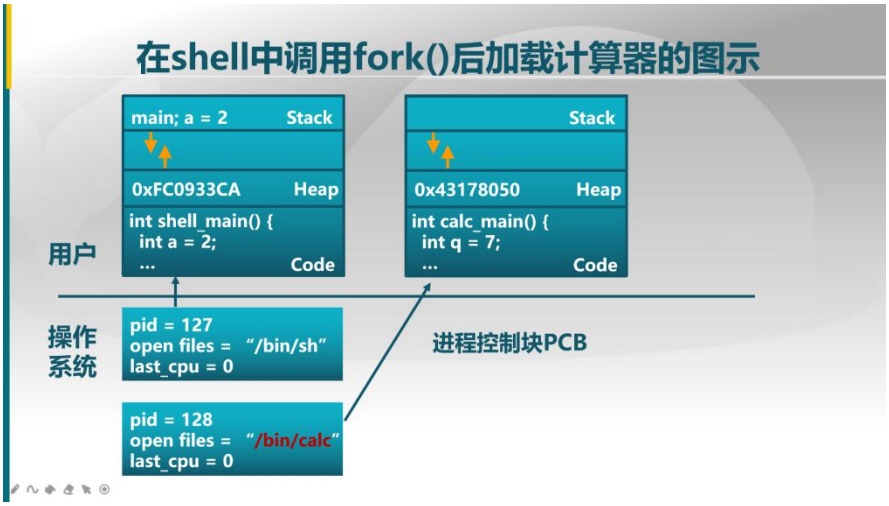
跳进 else，子进程(变量)pid 因为为 0，所以执行 exec,将加载的文件换掉，进而将子进程地址空间中的代码换掉，如下图所示：



从完整地址空间的角度来看，fork()后父进程和子进程的代码、堆(heap)及栈(stack)是一样的，只有 pid 不同：



当子进程执行 exec()后，会把整个子进程的地址空间内容换掉，变为：



下面看一个复杂些的例子：

fork()使用示例

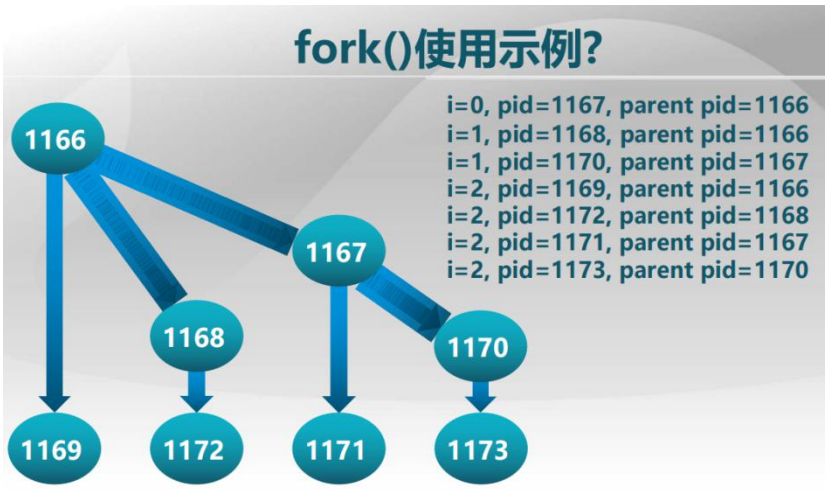
```
int main()
{
    pid_t pid;
    int i;

    for (i=0; i<LOOP; i++)
    {
        /* fork another process */
        pid = fork();
        if (pid < 0) { /*error occurred */
            fprintf(stderr, "Fork Failed" );
            exit(-1);
        }
        else if (pid == 0) { /* child process */
            fprintf(stdout, "i=%d, pid=%d, parent pid=%d\n" ,I,
                    getpid() ,getppid());
        }
    }
    wait(NULL);
    exit(0);
}
```

循环内部有一个 fork,假定该循环循环三次，问题：此时 fork()会被执行几次？

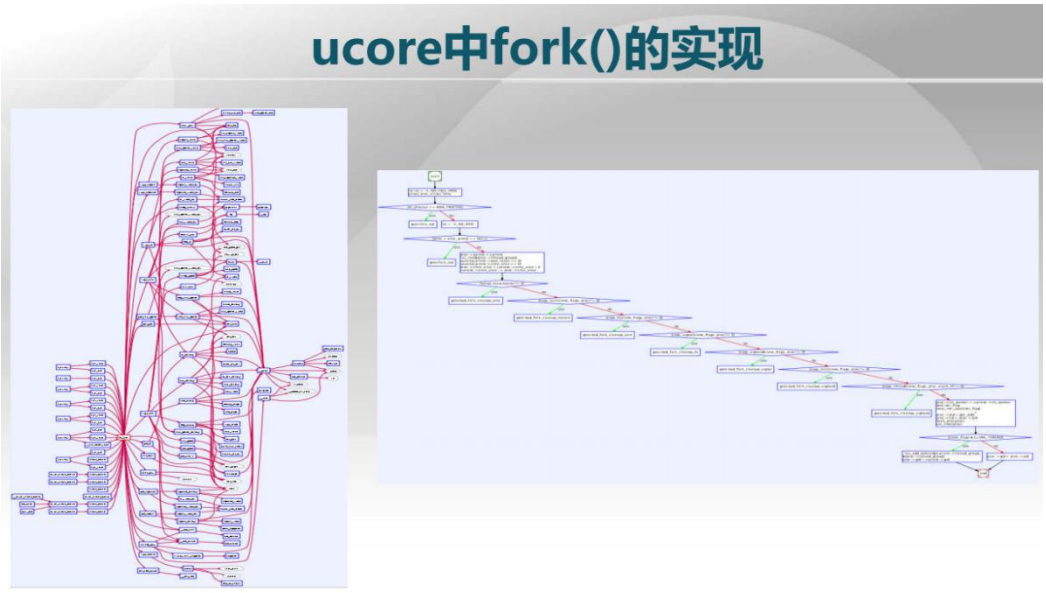
首先 if(pid<0)是检查 fork 是否成功，不成功直接退出，若成功，在子进程中打印一行信息，然后 wait(NULL),即等待它(子进程)的子进程结束，最后整个进程结束。

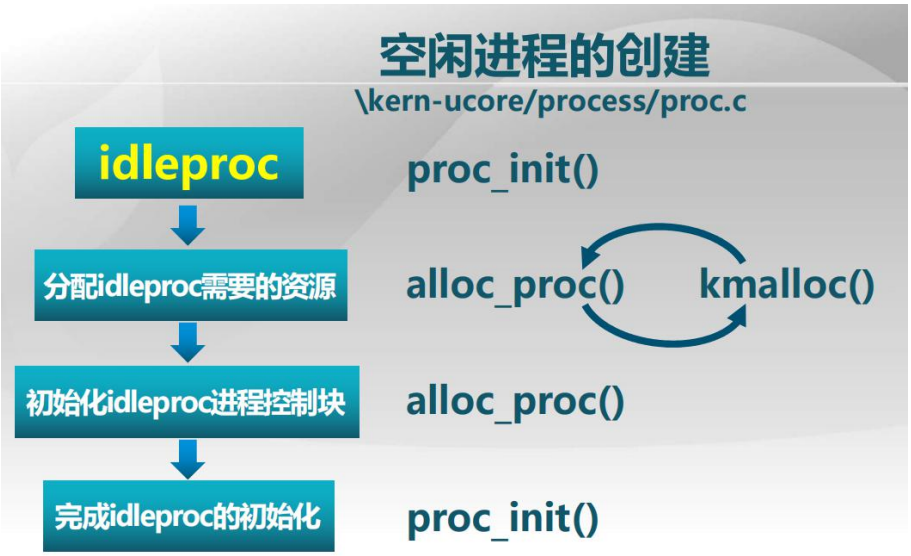
下面是该进程 fork 的过程：



注意，fork()完后，父进程和子进程的指令指针都在 if(pid<0)处。右侧具体的输出顺序会根据就绪队列的调度算法而有所变化。

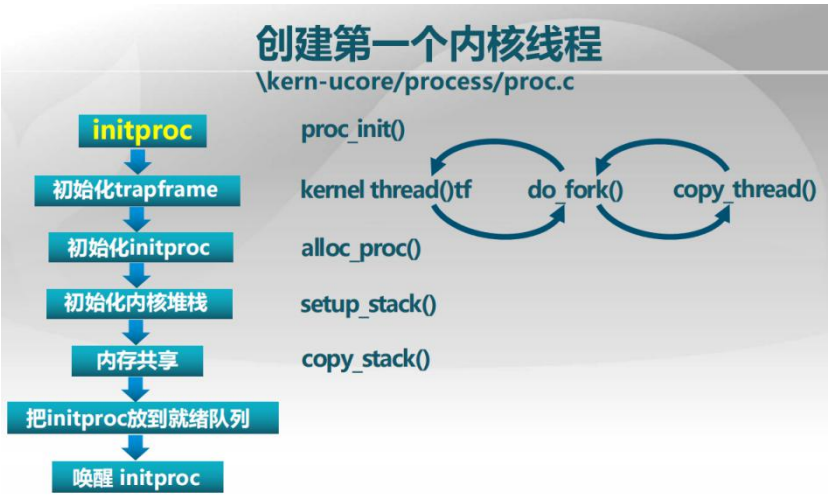
下面看实际系统 ucore 怎么做。

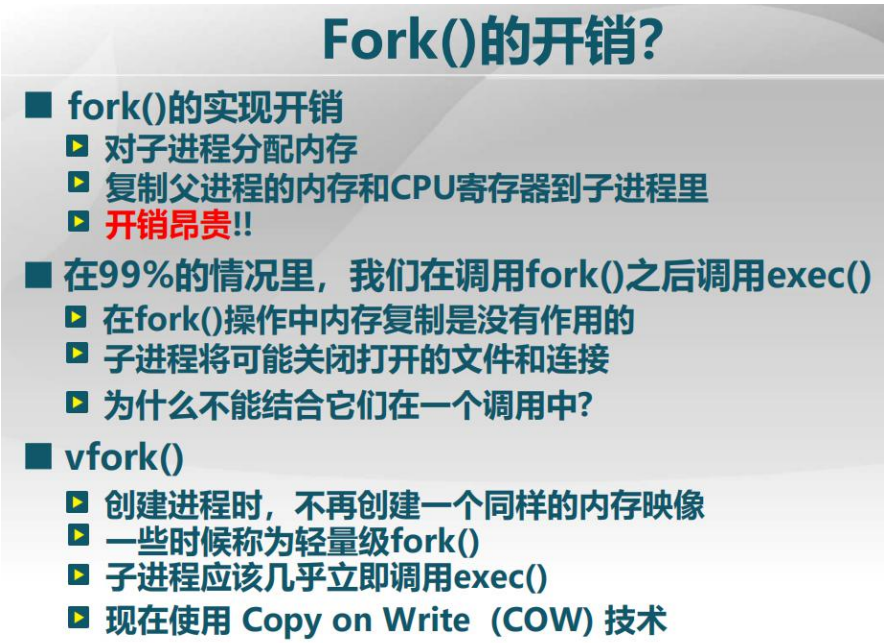




若用户代码都执行完，系统没有新的任务要执行，这时系统处于暂停状态，但实际上这时 CPU 并没有完全停下来，CPU 实际上还在执行指令，这时执行的实际上就是空闲进程的处理。Kmalloc 是分配存储资源的。其在 ucore 首先从 init.c 文件的 kern_init 开始内核的初始化，其中有 proc_init()，是页表的初始化，其具体定义见 proc.c.

另外是初始化进程，在用户态进程初始化之前，先创建了一个内核线程，即下图所示：





Cow:写时复制技术，任何一个进程创建时，都是在后面要用时，才延迟过来进行复制。如果直接是覆盖，复制就不会进行。

关于创建新进程的描述正确的是

A.fork() 创建子进程中，会复制父进程的所有变量和内存

B.子进程的 fork()返回

C.父进程的 fork()在创建子进程成功后，返回子进程标识符

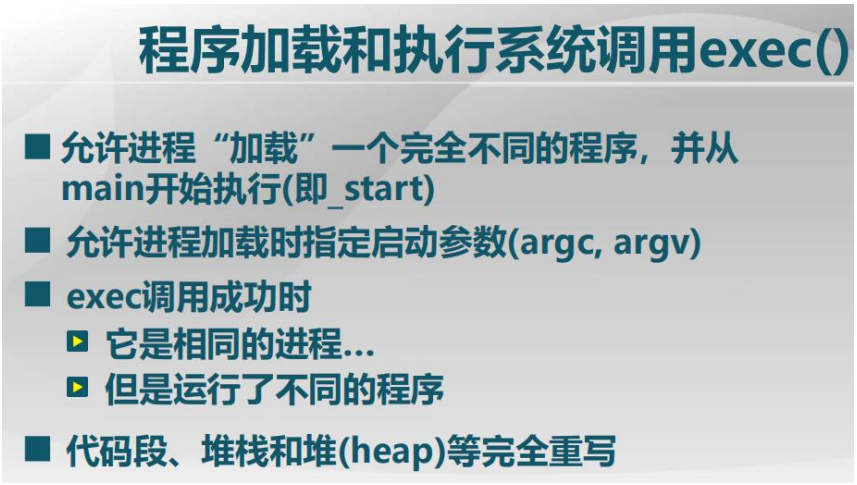
D.fork() 创建子进程中，会复制父进程的页表

都对，注意 A：fork() 创建一个继承的子进程，复制父进程的所有 CPU 寄存器(有一个寄存器例外，即 pid 的寄存器)，复制父进程的所有变量和内存。注意 A 不要漏选

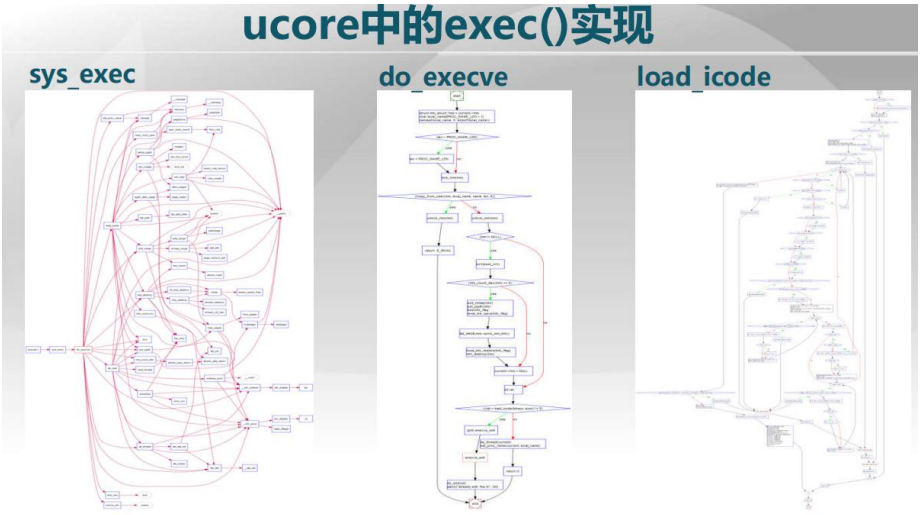
第三节 进程加载

进程加载指用户的应用程序通过系统调用 exec()加载，来完成对一个新的可执行文件的加载。用 exec()系统调用可以加载一个新的可执行文件到内存中覆盖原来的进程地址空间，然后开始执行。

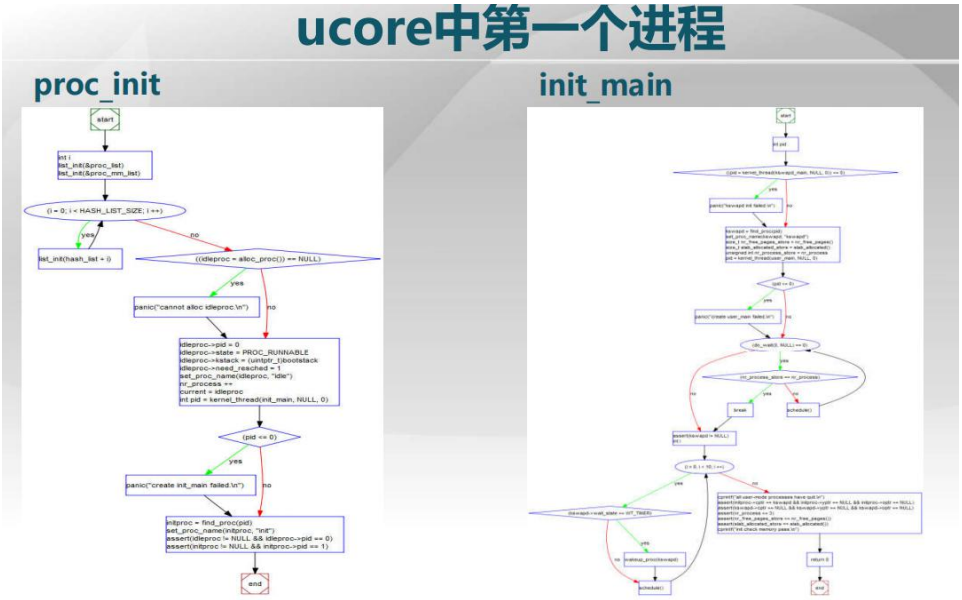
系统加载过程复习：CPU 加电->启动 BIOS 中的程序->硬盘上加载引导扇区->bootloader->内核映像->用户应用程序



加载时，有可执行文件格式的识别问题，不同系统，可加载的可执行文件的格式是不同的。



Exec()的实现即从外存把可执行文件加载进来，跳转到上面执行即可。这和引导扇区不同，引导扇区是读一块内容放在那里，对其格式没有任何理解(可理解为引导扇区读进来的东西格式单一，系统不需要去理解其格式？)。但是，执行一个应用程序时，可执行文件的格式是非常复杂的，所以 exec()主要工作就是可执行文件格式的识别。Sys_exec 获取进程相应参数，然后 do_execve，创建进程所需的内存相关的段结构，Load_icode 识别可执行文件的格式，并在内存里加载相应的段，然后执行。



Proc_init 中手工构造出一个进程控制块，然后把它放到就绪队列中，然后执行。这时启动的第一个用户态程序就是 init_main，这里就有 shell 程序。

关于进程加载执行的描述正确的是

A.系统调用 exec( )加载新程序取代当前运行进程

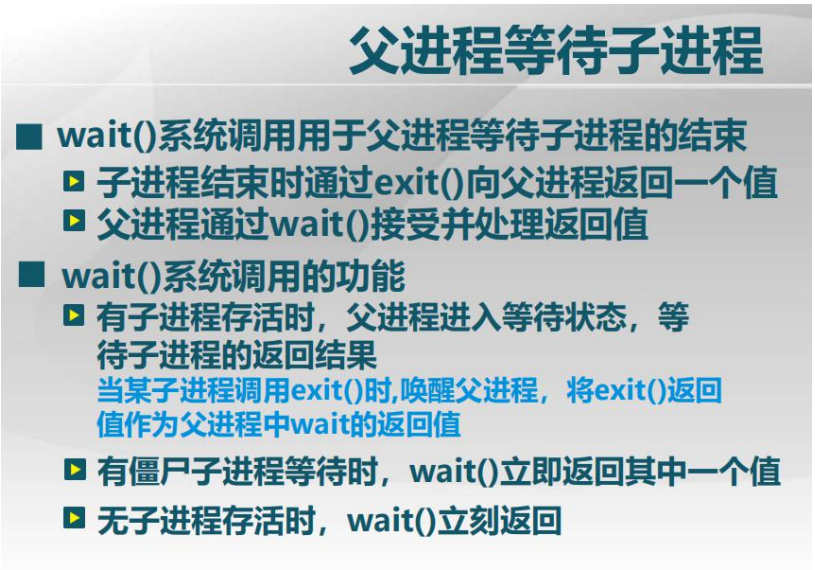
B.系统调用 exec( )允许进程“加载”一个完全不同的程序，并从 main 开始执行

C.exec 调用成功时，它是相同的进程，但是运行了不同的程序

D.exec 调用成功时，代码段、堆栈和堆(heap)等完全重写了

都对.注意 A 的意思应该是系统调用 exec( )加载新程序取代当前运行进程(中运行的内容)，不要漏选。

第四节 进程等待与退出



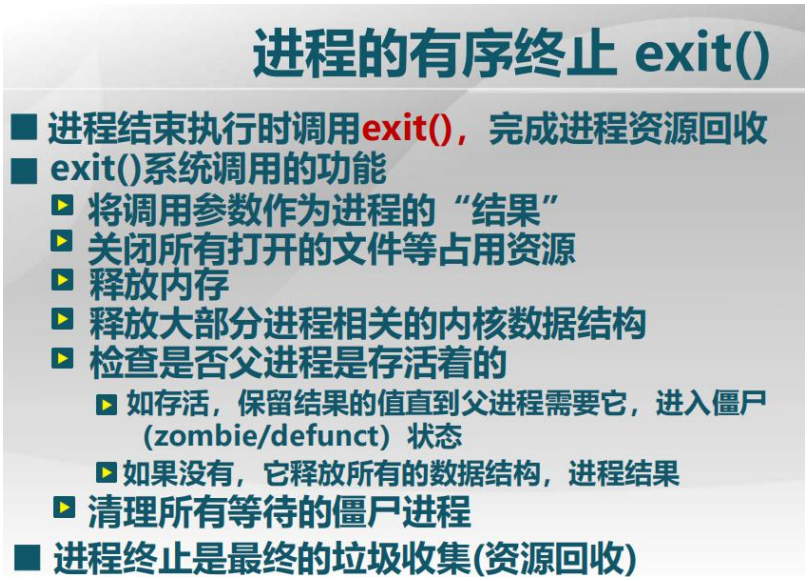
问题：wait()系统调用用于父进程等待子进程的结束，子进程结束时通过 exit()向父进程

返回一个值，父进程通过 wait()接受并处理返回值。则 wait()与 exit()谁先谁后？

先执行 wait()，后执行 exit()的情况：有子进程存活时，父进程进入等待状态，等待子进程的返回结果，当某子进程调用 exit()时,唤醒父进程，将 exit()返回值作为父进程中 wait 的返回值。

先执行 Exit()，后执行 wait()的情况：有僵尸子进程等待时，wait()立即返回其中一个值。

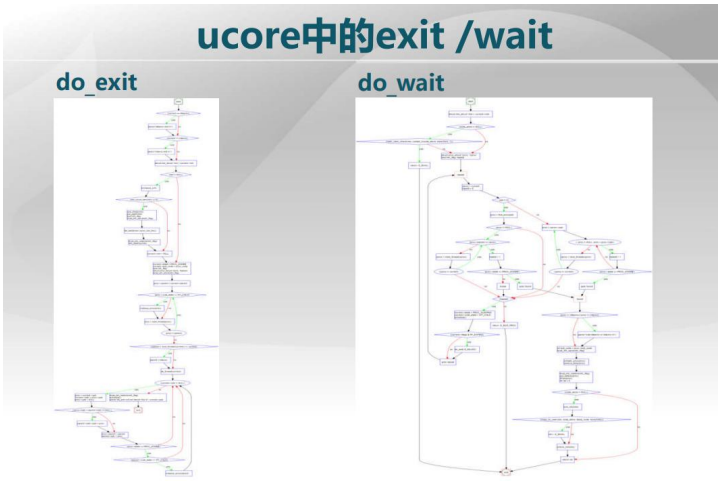
只执行 wait():无子进程存活时，wait()立刻返回.



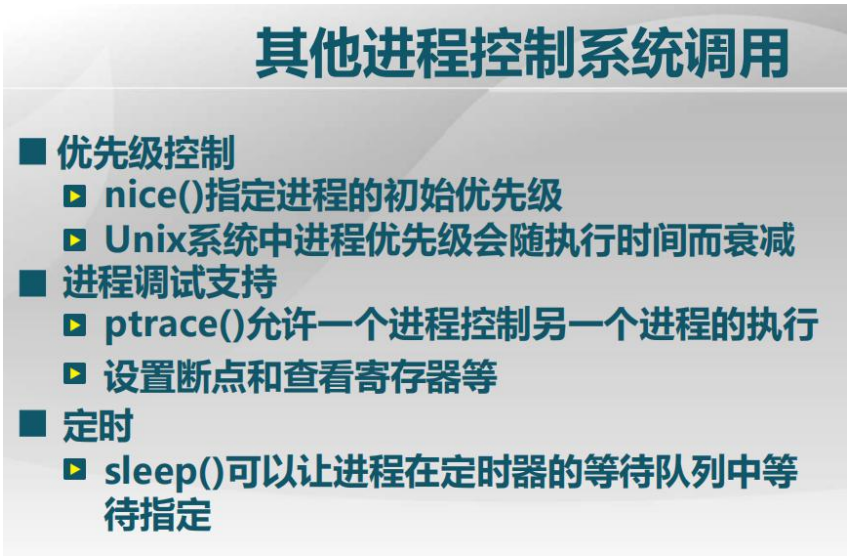
Exit()即有序退出，①其将调用参数作为父进程的结果，如某进程调用编译，编译成功返回调用参数 0，否则返回错误码，父进程可根据返回值做后续处理。

②资源回收：关闭所有打开的文件等占用资源，释放内存，释放大部分进程相关的内核数据结构。

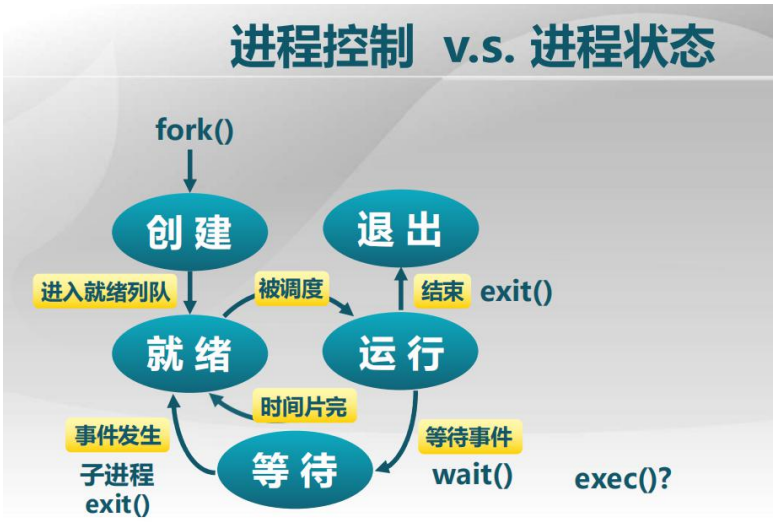
③子进程也会检查父进程：检查是否父进程是存活着的。如存活，保留结果的值直到父进程需要它，进入僵尸（zombie/defunct）状态。如果没有，它释放所有的数据结构，进程结果。



除创建 fork、加载 exec、等待 wait、退出 exit 这四个系统调用之外，还有一些其它系统调用：



Sleep()完成后会进入就绪状态。



子进程的 exit()会导致由于 wait()进入等待状态的父进程进入就绪状态。
Exec()是在运行过程中的一种状态。

有关管理进程等待的描述正确的是

A.wait()系统调用用于父进程等待子进程的结束

B.子进程结束时通过 exit()向父进程返回一个值

C.当某子进程调用 exit()时,唤醒父进程，将 exit()返回值作为父进程中 wait 的返回值

D.进程结束执行时调用 exit()，完成进程的部分占用资源的回收

都对，注意都对，D 项中，若父进程存活，exit()后子进程会进入僵尸队列中，而不会完成全部资源的回收。