Lab-5

实验目的：

- 了解第一个用户进程创建过程
- 了解系统调用框架的实现机制
- 了解ucore如何实现系统调用sys_fork/sys_exec/sys_exit/sys_wait来进行进程管理

实验内容：

练习0：填写已有实验

已填写

练习1：加载应用程序并执行（需要编程）

题目1：`do_execve`函数调用`load_icode`（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序。你需要补充`load_icode`的第6步，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好`proc_struct`结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

- 为了让用户级别的进程可以从内核返回到用户模式，所以：
  - `tf->gpr.sp` 应该是用户栈的顶部（即 sp 的值）。
  - `tf->status` 应该适用于用户程序（即 sstatus 的值）。
  - `tf->epc` 应该是用户程序的入口点（即 sepc 的值）。

代码如下：

```
tf->gpr.sp=USTACKTOP;
tf->status=sstatus&(~(SSTATUS_SPP|SSTATUS_SPIE));
tf->epc=elf->e_entry;
```

即：

- 将 tf 结构中的通用寄存器（gpr）中的栈指针（sp）设置为 USTACKTOP。这表明将用户栈的顶部地址赋给用户进程的栈指针。
- 将 tf 结构中的状态寄存器（status）设置为给定的 sstatus，但清除了 SPP（Supervisor Previous Privilege）和 SPIE（Supervisor Previous Interrupt Enable）标志。这两个标志通常用于处理从内核返回用户模式时的特权级别和中断使能状态。
- 将 tf 结构中的程序计数器（epc）设置为 ELF 文件的入口点地址。这是用户程序的启动地址，将控制权转移到用户程序的执行起点。

题目2：请简要描述这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

这其实就是`do_execve`函数的工作过程：

- 调用schedule函数，调度器占用了CPU的资源之后，用户态进程调用了exec系统调用，从而转入到了系统调用的处理过程，将控制权转移到了`syscall.c`中的`syscall`函数，然后根据系统调用号

转移给了`sys_exec`函数，在该函数中调用了`do_execve`函数来完成指定应用程序的加载。

- 在`do_execve`函数中，先获取当前进程的内存管理结构，将当前进程的页表换用内核页表之后，调用`load_icode`函数给用户进程建立一个能够让用户进程正常运行的用户环境，即以下过程：
- 调用mm_create函数来申请进程的内存管理数据结构mm所需内存空间，并对mm进行初始化。
- 调用setup_pgdir来申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表（boot_pgdir所指）的内容拷贝到此新目录表中，最后mm->pgdir指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核。
- 根据应用程序执行码的起始位置来解析此ELF格式的执行程序，并调用mm_map函数根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把vma插入到mm结构中，从而表明了用户进程的合法用户态虚拟地址空间。
- 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；
- 需要给用户进程设置用户栈，为此调用mm_mmap函数建立用户栈的vma结构，明确用户栈的位置在用户虚空间的顶端，大小为256个页，即1MB，并分配一定数量的物理内存且建立好栈的虚地址<->物理地址映射关系；
- 至此,进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被hello的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；
- 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让CPU转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。

练习2: 父进程复制自己的内存空间给子进程（需要编码）

题目1：创建子进程的函数`do_fork`在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过`copy_range`函数（位于kern/mm/pmm.c中）实现的。

do_fork函数调用copy_mm函数实现进程mm的复制，后者根据clone_flags & CLONE_VM的取值调用了dup_mmap函数。

dup_mmap函数在两个进程之间复制内存映射关系。具体来说，该函数的两个参数分别表示目标进程和源进程的内存管理结构mm。然后通过循环迭代，每次创建一个新的内存映射区域（vma），然后将其插入到目标进程的mm中，之后调用copy_range函数将源进程的内存映射区域的内容复制到目标进程中。

填写copy_range函数后，其代码实现如下：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do {
        // call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        // call get_pte to find process B's pte according to the addr start. If
        // pte is NULL, just alloc a PT
        if (*ptep & PTE_V) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
```

```
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            // get page from ptep
            struct Page *page = pte2page(*ptep);
            // alloc a page for process B
            struct Page *npage = alloc_page();
            assert(page != NULL);
            assert(npage != NULL);
            int ret = 0;
            /* LAB5:EXERCISE2 YOUR CODE
             * replicate content of page to npage, build the map of phy addr of
             * nage with the linear addr start
             *
             * Some Useful MACROs and DEFINEs, you can use them in below
             * implementation.
             * MACROs or Functions:
             *    page2kva(struct Page *page): return the kernel vritual addr of
             * memory which page managed (SEE pmm.h)
             *    page_insert: build the map of phy addr of an Page with the
             * linear addr la
             *    memcpy: typical memory copy function
             *
             * (1) find src_kvaddr: the kernel virtual address of page
             * (2) find dst_kvaddr: the kernel virtual address of npage
             * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
             * (4) build the map of phy addr of  nage with the linear addr start
             */
            void *src_kvaddr = page2kva(page);
            void *dst_kvaddr = page2kva(npage);
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
            ret = page_insert(to, npage, start, perm);
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

函数首先通过断言确保start和end是页面对齐且属于用户地址空间的。然后通过循环每次处理一页数据。在每次迭代中，函数调用 get_pte 函数查找源进程 A 的页表项（pte），如果不存在，则跳过当前页并继续下一个页面。如果 pte 存在，则再调用 get_pte 函数查找目标进程 B 的页表项（nptep）。如果 nptep 不存在，则分配一个新的页表并建立映射。

接下来，函数调用 alloc_page 函数为进程 B 分配一个新的物理页面（npage）。然后，函数将源进程 A 的物理页面（page）中的内容复制到新的物理页面中。具体实现上，就是通过page2kva函数获取到目标进程和源进程各自页面的虚拟地址，然后使用memcpy函数实现复制。页面复制完成后，函数再调用 page_insert 函数将新的物理页面与目标进程 B 的线性地址建立映射。

题目2：如何设计实现 `Copy on Write` 机制？给出概要设计，鼓励给出详细设计。

Copy on Write（COW）机制是一种在多进程共享资源时，延迟复制的技术。它允许多个进程共享同一块内存空间，直到其中一个进程尝试修改数据时，才会进行实际的复制操作，以确保数据的独立性。

在设计该机制时，考虑到了以下两点：

- 未修改时，如何实现内存共享？
- 修改时，如何实现资源复制以及之后的数据一致性？

针对以上两点，进行了如下设计：

- 内存管理：
  - 使用页表来跟踪进程间共享的内存页，记录每个页的引用计数。
  - 当进程创建时，内核分配一块共享内存区域，并将该区域映射到多个进程的地址空间中。
- 写时复制实现：
  - 当进程试图写入一个共享页时，内核会触发页故障：
    - 检查被访问的页是否是共享的。
    - 如果引用计数为1，表示该页只有一个进程在使用，直接允许修改。
    - 如果引用计数大于1，执行复制操作：
      - 分配一个新的页，并将原始页的内容复制到新页中。
      - 更新进程的页表，使其指向新的页。
      - 更新引用计数和页表条目，确保进程间的独立性。
- 引用计数：
  - 引用计数用于跟踪共享页的引用情况。每当一个新的进程开始使用某个共享页时，引用计数增加；当进程停止使用该页时，引用计数减少。
  - 当引用计数为0时，释放对应的页。
- 数据一致性：
  - 确保在进行写时复制时，修改后的页仅影响到修改它的进程，而不影响其他进程。
  - 维护页表和引用计数，以确保每个进程都能正确访问到自己的数据，并且在修改时能够获得独立的拷贝。
- 同步机制：
  - 在进行写时复制时，需要采取同步机制以避免多个进程同时修改相同的共享页。可以使用锁或其他同步手段来确保数据的一致性和正确性。

练习3：阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

- 简要对fork/exec/wait/exit函数进行分析

①fork：执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程

```
// 调用过程：fork->SYS_fork->do_fork + wakeup_proc

// wakeup_proc 函数主要是将进程的状态设置为等待。

// do_fork()
1、分配并初始化进程控制块(alloc_proc 函数);
2、分配并初始化内核栈(setup_stack 函数);
3、根据 clone_flag标志复制或共享进程内存管理结构(copy_mm 函数);
4、设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文(copy_thread 函数);
5、把设置好的进程控制块放入hash_list 和 proc_list 两个全局进程链表中;
6、自此,进程已经准备好执行了,把进程状态设置为“就绪”态;
7、设置返回码为子进程的 id 号。
```

- 用户态程序调用sys_fork()系统调用，通过syscall进入内核态。
- 内核态处理sys_fork()系统调用，调用do_fork()函数创建子进程，完成后返回到用户态。

②exit：会把一个退出码error_code传递给ucore，ucore通过执行内核函数do_exit来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作。

```
// 调用过程： SYS_exit->exit

1、先判断是否是用户进程，如果是，则开始回收此用户进程所占用的用户态虚拟内存空间;（具体的回收过程不作详细说明）
2、设置当前进程的中hi性状态为PROC_ZOMBIE，然后设置当前进程的退出码为error_code。表明此时这个进程已经无法再被调度了，只能等待父进程来完成最后的回收工作（主要是回收该子进程的内核栈、进程控制块）
3、如果当前父进程已经处于等待子进程的状态，即父进程的wait_state被置为WT_CHILD，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。
4、如果当前进程还有子进程,则需要把这些子进程的父进程指针设置为内核线程init,且各个子进程指针需要插入到init的子进程链表中。如果某个子进程的执行状态是 PROC_ZOMBIE,则需要唤醒 init来完成对此子进程的最后回收工作。
5、执行schedule()调度函数，选择新的进程执行。
```

- 用户态程序调用sys_exit()系统调用，通过syscall进入内核态。
- 内核态处理sys_exit()系统调用，调用do_exit()函数结束当前进程，最终返回到用户态。

③execve：完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。

```
// 调用过程： SYS_exec->do_execve

1、首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。
2、接下来是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。之后就是调用load_icode从而使之准备好执行。
```

- 用户态程序调用sys_exec()系统调用，通过syscall进入内核态。
- 内核态处理sys_exec()系统调用，调用do_execve()函数加载新的程序，但由于当前是在S mode下，无法直接进行上下文切换。因此使用ebreak产生断点中断，转发到syscall()函数，在该函数 中完成上下文切换，最终返回到用户态。

④wait：等待任意子进程的结束通知。wait_pid函数等待进程id号为pid的子进程结束通知。这两个函数最终访问sys_wait系统调用接口让ucore来完成对子进程的最后回收工作。

```
// 调用过程： SYS_wait->do_wait

1、 如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程;
2、 如果此子进程的执行状态不为PROC_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC_SLEEPING（睡眠），睡眠原因为WT_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行;
3、 如果此子进程的执行状态为 PROC_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc_list和hash_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。
```

- 用户态程序调用sys_wait()系统调用，通过syscall进入内核态。
- 内核态处理sys_wait()系统调用，调用do_wait()函数等待子进程退出，完成后返回到用户态。

-

请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）

```
process state    :   meaning        -- reason
-----------------------------------------------------------------------------

  PROC_UNINIT    :  uninitialized      -- alloc_proc

  PROC_SLEEPING  :  sleeping           -- try_free_pages, do_wait, do_sleep

  PROC_RUNNABLE  :  runnable(maybe running) -- proc_init, wakeup_proc,

  PROC_ZOMBIE    :  almost dead        -- do_exit
```

扩展练习 Challenge

题目1：

接下来将说明如何实现“Copy on Write”机制，该机制的主要思想为使得进程执行fork系统调用进行复制的时候，父进程不会简单地将整个内存中的内容复制给子进程，而是暂时共享相同的物理内存页；而当其中一个进程需要对内存进行修改的时候，再额外创建一个自己私有的物理内存页，将共享的内容复制过去，然后在自己的内存页中进行修改；根据上述分析，主要对实验框架的修改应当主要有两个部分，一个部分在于进行fork操作的时候不直接复制内存，另外一个处理在于出现了内存页访问异常的时候，会将共享的内存页复制一份，然后在新的内存页进行修改，具体的修改部分如下：

- do fork部分：在进行内存复制的部分，比如copy_range函数内部，不实际进行内存的复制，而是将子进程和父进程的虚拟页映射上同一个物理页面，然后在分别在这两个进程的虚拟页对应的PTE部分将这个页置成是不可写的，同时利用PTE中的保留位将这个页设置成共享的页面，这样的话如果应用程序试图写某一个共享页就会产生页访问异常，从而可以将控制权交给操作系统进行处理；
- page fault部分：在page fault的ISR部分，新增加对当前的异常是否由于尝试写了某一个共享页面引起的，如果是的话，额外申请分配一个物理页面，然后将当前的共享页的内容复制过去，建立出错的线性地址与新创建的物理页面的映射关系，将PTE设置设置成非共享的；然后查询原先共享的物理页面是否还是由多个其他进程共享使用的，如果不是的话，就将对应的虚地址的PTE进行修改，删掉共享标记，恢复写标记；这样的话page fault返回之后就可以正常完成对虚拟内存（原想的共享内存）的写操作了；
- 上述实现有一个较小的缺陷，在于在do fork的时候需要修改所有的PTE，会有一定的时间效率上的损失；可以考虑将共享的标记加在PDE上，然后一旦访问了这个PDE之后再将标记下传给对应的PTE，这样的话就起到了标记延迟和潜在的标记合并的左右，有利于提升时间效率；

`copy_range()`：

```
int
copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    // 0x00200000 ~ 0xB0000000
```

```
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do {
        //call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //call get_pte to find process B's pte according to the addr start. If
pte is NULL, just alloc a PT
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            //get page from ptep
            struct Page *page = pte2page(*ptep);
            // alloc a page for process B
            struct Page *npage=alloc_page();
            assert(page!=NULL);
            assert(npage!=NULL);
            int ret=0;
            if (share) {
                cprintf("Sharing the page 0x%x\n", page2kva(page));
                // 物理页面共享，并设置两个PTE上的标志位为只读
                page_insert(from, page, start, perm & ~PTE_W);
                ret = page_insert(to, page, start, perm & ~PTE_W);
            } else {
                // (1) find src_kvaddr: the kernel virtual address of page
                void *src_kvaddr = page2kva(page);
                // (2) find dst_kvaddr: the kernel virtual address of npage
                void *dst_kvaddr = page2kva(npage);
                // (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
                memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
                // (4) build the map of phy addr of  nage with the linear addr
start
                // 将该页面设置至对应的PTE中
                ret = page_insert(to, npage, start, perm);
            }

            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

`do_pgfault`:

```
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    // 页访问异常错误码有32位。
    // 位0为 1 表示对应物理页不存在；
    // 位 1 为 1 表示写异常（比如写了只读页；位 2 为 1 表示访问权限异常（比如用户态程序访问内核空间
的数据）
```

```
    // 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。
    // CR2用于发生页异常时报告出错信息。当发生页异常时，处理器把引起页异常的线性地址保存在CR2
中。
    // 操作系统中对应的中断服务例程可以检查CR2的内容，从而查出线性地址空间中的哪个页引起本次异
常。
    int ret = -E_INVAL;

    //try to find a vma which include addr
    // 根据addr从vma中查找对应的vma_struct
    DEBUG("mm = [%p], error_code = [%x], addr = [%x]", mm, error_code, addr);
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++;
    //If the addr is in the range of a mm's vma?
    DEBUG("mm = [%p], vma = [%p], addr = [%x]\n", mm, vma, addr);
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    //check the error_code
    // 与 00000011
    //
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
    case 2: /* error code flag : (W/R=1, P=0): write, not present */
        if (!(vma->vm_flags & VM_WRITE)) {
            cprintf("do_pgfault failed: error code flag = write AND not present,
but the addr's vma cannot write\n");
            goto failed;
        }
        break;
    case 1: /* error code flag : (W/R=0, P=1): read, present */
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            cprintf("do_pgfault failed: error code flag = read AND not present,
but the addr's vma cannot read or exec\n");
            goto failed;
        }
    }
    /* IF (write an existed addr ) OR
     *    (write an non_existed addr && addr is writable) OR
     *    (read  an non_existed addr && addr is readable)
     * THEN
     *    continue process
     */
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
    // 根据get_pte来获取pte如果不存在，则分配一个新的
```

```
    ptep = get_pte(mm->pgdir, addr, 1);
    if (ptep == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }
    struct Page *p;
    // 如果对应的物理页不存在，分配一个新的页，且把物理地址和逻辑地址映射
    if (*ptep == 0) {
        p = pgdir_alloc_page(mm->pgdir, addr, perm);
        if (p == NULL) {
            cprintf("alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {
        struct Page *page=NULL;
        // 如果当前页错误的原因是写入了只读页面
        if (*ptep & PTE_P) {
            // 写时复制：复制一块内存给当前进程
            cprintf("\n\nCOW: ptep 0x%x, pte 0x%x\n",ptep, *ptep);
            // 原先所使用的只读物理页
            page = pte2page(*ptep);
            // 如果该物理页面被多个进程引用
            if (page_ref(page) > 1) {
                struct Page* newPage = pgdir_alloc_page(mm->pgdir, addr, perm);
                void * kva_src = page2kva(page);
                void * kva_dest = page2kva(newPage);
                memcpy(kva_dest, kva_src, PGSIZE);
            } else {
                page_insert(mm->pgdir, pa2page, addr, perm);
            }
        }

        // 如果不全为0，则可能被交换到了swap磁盘中
        if(swap_init_ok) {
            struct Page *page=NULL;
            int swapIn;
            // 从磁盘中换出
            swapIn = swap_in(mm, addr, &page);
            if (swapIn != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }

            // build the map of phy addr of an Page with the linear addr la
            page_insert(mm->pgdir, page, addr, perm);

            swap_map_swappable(mm, addr, page, 1);
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok, but ptep is %x, failed\n", *ptep);
            goto failed;
        }
    }
   ret = 0;
failed:
    return ret;
}
```

题目2：说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

本次实验中，用户程序是在执行 `execve` 系统调用时被预先加载到内存中的。与我们常用的操作系统加载过程的区别主要在于执行上述加载步骤的时机。在常用的操作系统中，用户程序通常在运行时（runtime）才被加载到内存。当用户启动程序或运行可执行文件时，操作系统负责将程序从磁盘加载到内存，然后执行。这么做的原因是**简化用户程序执行过程：**预先加载用户程序可以简化用户程序的执行过程，使得执行过程更加直接和快速。

知识点总结

实验执行流程概述

1. 编译器编译用户程序源代码为可执行的目标程序（包含执行代码&内存分配信息）
2. 根据分配信息为其分配内存，并将其加载进内存，为其建立进程（分配资源等）
3. 通过进程调度运行该程序

系统调用

用户程序在用户态运行(U mode), 系统调用在内核态执行(S mode)。涉及CPU特权级切换的过程，`ecall` 可以从U到S。S mode调用 `OpenSBI` 提供的 `M mode` 接口。用 `ecall` 调用进入M mode，剩下的事情交给底层的OpenSBI来完成。

内核线程到用户进程

各个内核线程间的配合是较为协调的，能够相互考虑各自的资源利用情况，从而可以在适当的时候使用或不使用资源。而用户进程则相对利己，只考虑自身的资源利用情况，所以需要操作系统管理它们，让有效的资源得到协调而安全的利用。

让用户进程正常运行的用户环境

用户环境由以下部分组成

- 建立用户虚拟空间的页表和支持页换入换出机制的用户内存访存错误异常服务例程：提供地址隔离和超过物理空间大小的虚存空间。
- 应用程序执行的用户态 CPU 特权级：在用户态 CPU 特权级，应用程序只能执行一般指令，如果特权指令，结果不是无效就是产生“执行非法指令”异常；
- 系统调用机制：给用户进程提供“服务窗口”；
- 中断响应机制：给用户进程设置“中断窗口”，这样产生中断后，当前执行的用户进程将被强制打断，CPU 控制权将被操作系统的中断服务例程使用。

用户态进程的执行过程

用户进程执行过程中，会发生系统调用、外部中断等造成特权级切换的情况，这时执行的进程到底是什么级别的进程呢？

从进程控制块（PCB）的角度理解，当PCB发生变化时、即执行了进程执行现场的切换，就认为是进程的分界点，因此可以把执行应用程序的代码一直到执行操作系统中的进程切换处为止都认为是一个应用程序的执行过程。

从指令执行/逻辑功能的角度理解，就需要进一步的分类讨论。对于用户进程主动的切换行为（如系统调用），进程代码包括用户和内核两部分。而对于用户进程被动的切换行为（如外设中断和CPU执行异常），切换后的代码逻辑对用户程序而言不可见，因此是另外一段执行逻辑。

5状态模型

- 创建态：创建了进程控制块（PCB）时，但还未分配任何资源，之后会转变为就绪态
- 就绪态：对进程分配/准备了运行所需的各种资源（虚拟内存空间、可执行代码、所需数据等），但还未分配CPU资源，之后会转变为运行态
- 运行态：进程调度后实现了上CPU运行。之后会出现三种情况，分别对应切换到三种状态——等待某事件（阻塞态）、可用时间结束（就绪态）、程序运行结束（退出态）
- 阻塞态：由于需要等待某事件（硬件资源可用、某数据产生等）发生而进入的状态，之后会切换为就绪态
- 退出态：回收进程的各种资源，最后回收对应的进程控制块