中国科学院大学
University of Chinese Academy of Sciences

进程

中国科学院大学计算机学院

2025-09-24

内容提要

- 进程的起源
- 进程的概念与表示
- 进程的状态
- 进程的操作原语（API）

进程的起源

- 从操作系统的发展看何时需要进程管理
  - 通用函数库
    - 用户按分配的时间段自行提交程序
  - 单道批处理系统
    - 支持多个程序一次提交（批处理）
    - 但任何时刻只运行一个程序（单道）
  - 多道批处理系统（multi-programming）
    - 支持多个程序同时运行
    - 多个程序共享使用CPU、内存等资源
  - Time-sharing Operating System
    - Compatible Time Sharing System (CTSS)，第一个通用的分时操作系统，允许运行中的程序通过时间片进行切换，“进程”开始登上历史舞台
    - Multics 1965、UNIX 1974

内容提要

- 进程的起源
- 进程的概念与表示
- 进程的状态
- 进程的操作原语（API）

进程的概念

- 进程是指一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程
- 进程刻画了一个程序运行所需要的资源和运行状态

程序运行时的资源使用视图

- 示例程序

```
#include <stdio.h>

void calculate()  {
    int  *array  =  malloc(5*sizeof(int));
    array[0]  =  1;
    for(int  i=1;i<5;i++) {
        array[i]  =  array[i-1] + i;
    }
}

int main(int argc, char *argv[]) {
    int fd = open("/tmp/examplefile", O_CREAT|O_WRONLY, S_IRWXU);
    calculate()
    write(fd, ”Finish\n", 7);
    close(fd);
    return 0;
}
```

内存

CPU

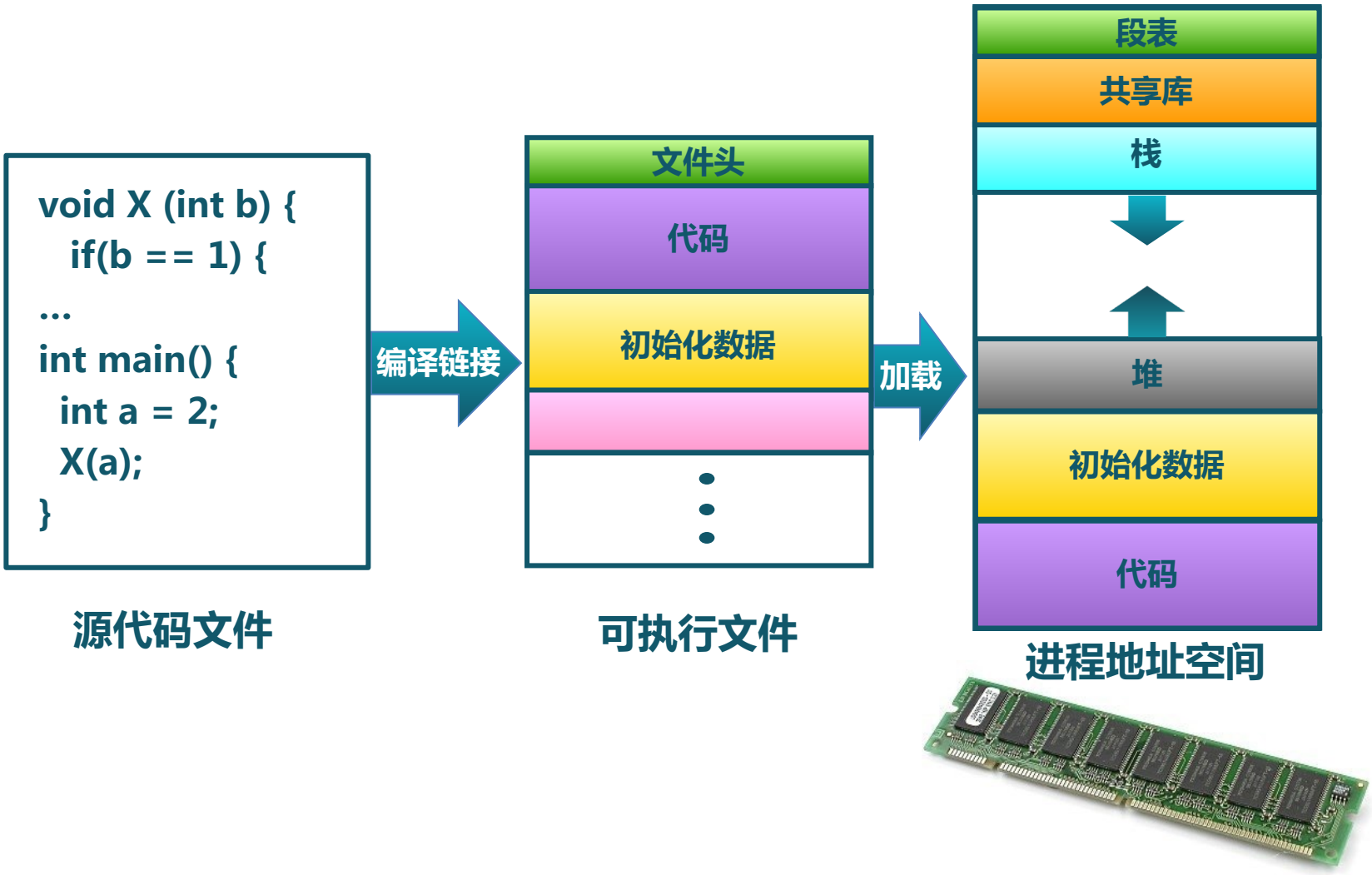
I/O设备

进程使用CPU资源

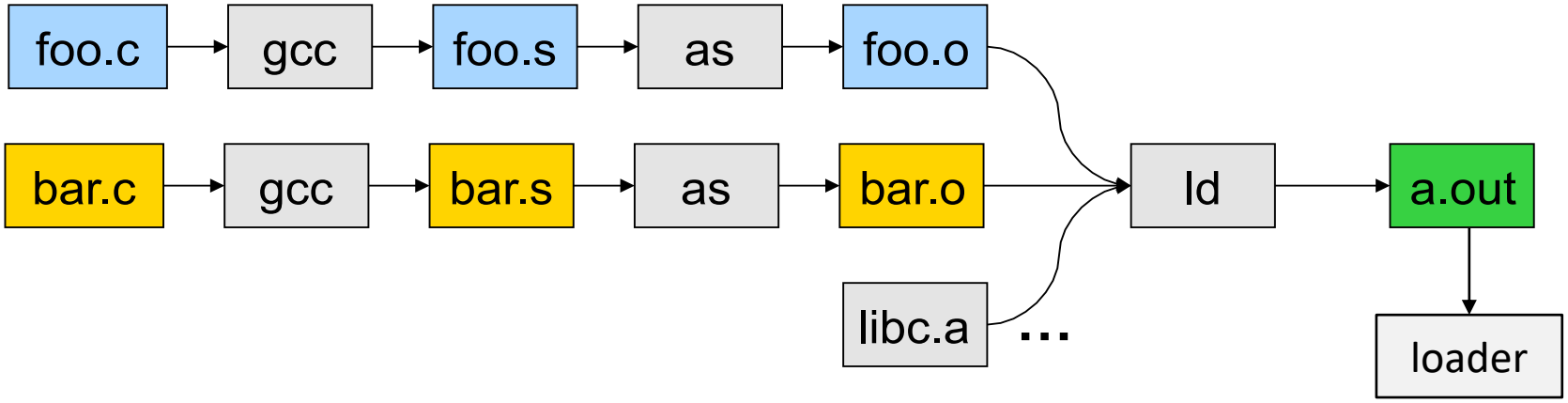
- 寄存器
  - 通用寄存器
  - Program Counter寄存器：存储下一条指令的地址
  - Stack Pointer寄存器：用作栈指针
  - Return Address寄存器：存储函数调用后的返回地址
  - 其他状态寄存器等
    - 例如CSR寄存器

进程使用内存资源

- 内存资源抽象为进程地址空间（Address Space）



从源码到可执行文件



- 编译器将源码程序编译成汇编文件
- 汇编器将汇编文件汇编为可重定位的目标文件
- 链接器将多个目标文件链接成一个可执行文件
- 加载器将可执行文件加载至内存中，开始执行程序

可执行文件的组成

- 示例程序
  - gcc examp.c -o example

```c
#include <stdlib.h>
#include <unistd.h>

int main() {
    while (1) {
        sleep(1000);
    }
    return 0;
}
```

可执行文件的组成

- readelf -l example
- 由Segments组成，其中LOAD类型Segment包含代码和数据

```
Elf 文件类型为 EXEC (可执行文件)
Entry point 0x4004a0
There are 9 program headers, starting at offset 64

程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000760 0x0000000000000760  R E    0x200000
  LOAD           0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000224 0x0000000000000228  RW     0x200000
  DYNAMIC        0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000638 0x0000000000400638 0x0000000000400638
                 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000200 0x0000000000000200  R      0x1

 Section to Segment mapping:
  段节...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
```

可执行文件的组成

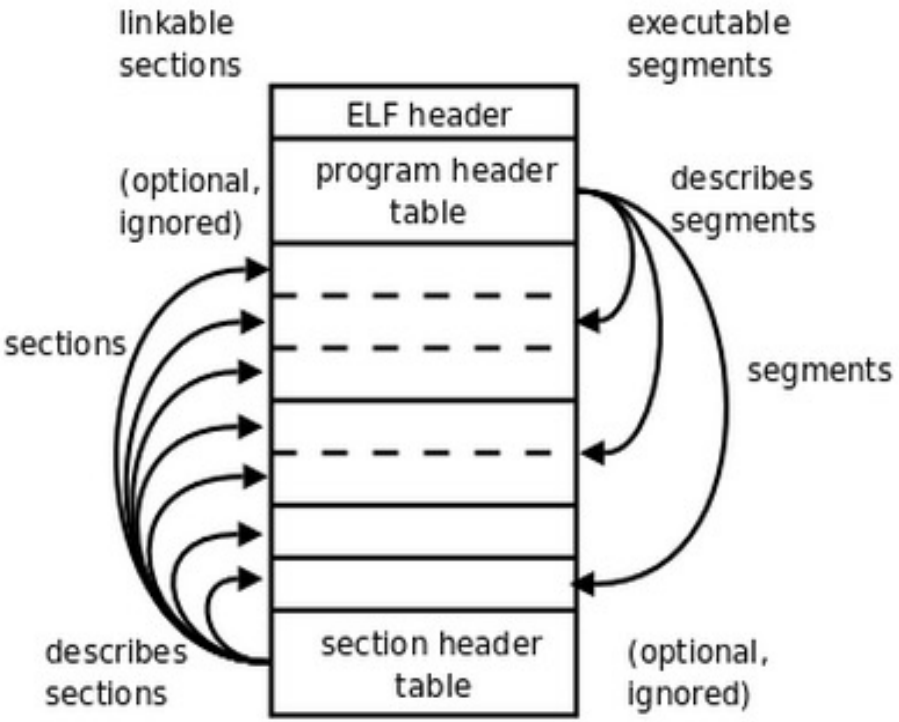
- readelf -S example
- 可执行文件同时包含链接视图，由sections组成

```
There are 30 section headers, starting at offset 0x2a50:

节头:
  [号] 名称              类型             地址              偏移量
       大小              全体大小         旗标   链接   信息   对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000090  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400348  00000348
       0000000000000074  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           00000000004003bc  000003bc
       000000000000000c  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          00000000004003c8  000003c8
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             00000000004003e8  000003e8
       0000000000000060  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000400448  00000448
       0000000000000018  0000000000000018  AI       5    22     8
  [11] .init             PROGBITS         0000000000400460  00000460
       000000000000001b  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000400480  00000480
       0000000000000020  0000000000000010  AX       0     0    16
  [13] .text             PROGBITS         00000000004004a0  000004a0
       0000000000000175  0000000000000000  AX       0     0    16
  [14] .fini             PROGBITS         0000000000400618  00000618
       000000000000000d  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000400628  00000628
       0000000000000010  0000000000000000   A       0     0     8
  [16] .eh_frame_hdr     PROGBITS         0000000000400638  00000638
       000000000000003c  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         0000000000400678  00000678
       00000000000000e8  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY       0000000000600e00  00000e00
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       0000000000600e08  00000e08
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .dynamic          DYNAMIC          0000000000600e10  00000e10
       00000000000001d0  0000000000000010  WA       6     0     8
  [21] .got              PROGBITS         0000000000600fe0  00000fe0
       0000000000000020  0000000000000008  WA       0     0     8
  [22] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000020  0000000000000008  WA       0     0     8
  [23] .data             PROGBITS         0000000000601020  00001020
       0000000000000004  0000000000000000  WA       0     0     1
  [24] .bss              NOBITS           0000000000601024  00001024
       0000000000000004  0000000000000000  WA       0     0     1
  [25] .comment          PROGBITS         0000000000000000  00001024
       000000000000002c  0000000000000001  MS       0     0     1
  [26] .gnu.build.attrib NOTE             0000000000a01028  00001050
       000000000000090c  0000000000000000           0     0     4
  [27] .symtab           SYMTAB           0000000000000000  00001960
       0000000000000990  0000000000000018          28    82     8
  [28] .strtab           STRTAB           0000000000000000  000022f0
       0000000000000645  0000000000000000           0     0     1
  [29] .shstrtab         STRTAB           0000000000000000  00002935
       0000000000000119  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

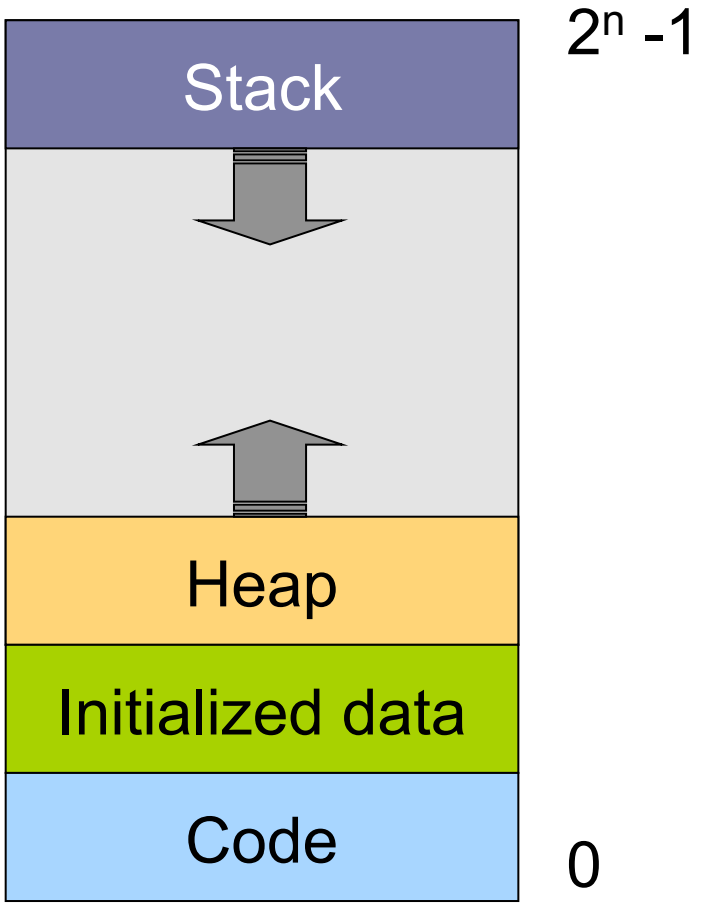
ELF文件

- ELF文件
  - Executable and Linking Format
  - Linux和Unix-like系统上的二进制格式标准，为不同类型文件提供了统一格式
    - 可重定位的目标文件(.o)
    - 可执行文件
    - 可被共享的文件(.so)
  - 链接视图
    - 由section（节）组成
  - 执行视图
    - 由segments（段）组成



进程地址空间的布局

- 组成内容
  - 代码段（.text)：指令序列
  - 数据段 (.data)：已经初始化的全局变量和静态变量
  - 未初始化数据段（.bss）：未初始化的全局变量和静态变量
  - 只读数据段（.rodata）：只读数据，例如字符串常量
  - 栈 (stack)
  - 堆 (heap)
- 布局特点
  - 分离代码和数据
  - 栈和堆相向增长

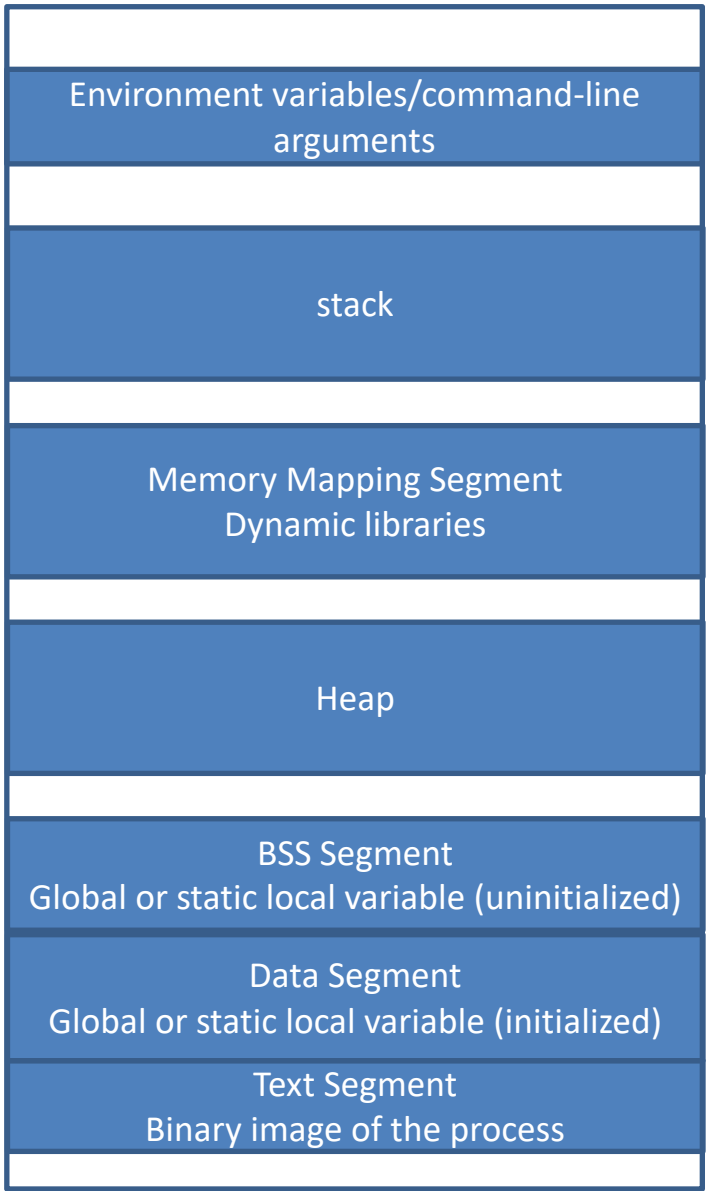


进程地址空间的布局

- 代码段/数据段
  - 链接器确定代码段和数据段（LOAD Segments）的内存地址（虚拟地址）
  - 加载器根据地址信息加载代码段和数据段至内存
- 栈
  - 进程创建时由操作系统动态确定栈的起始地址
- 堆
  - 链接器确定数据段结束地址（潜在的堆起始地址）
  - 加载器和操作系统动态确定堆的实际起始地址
  - 通过库函数malloc/free等进行分配和释放

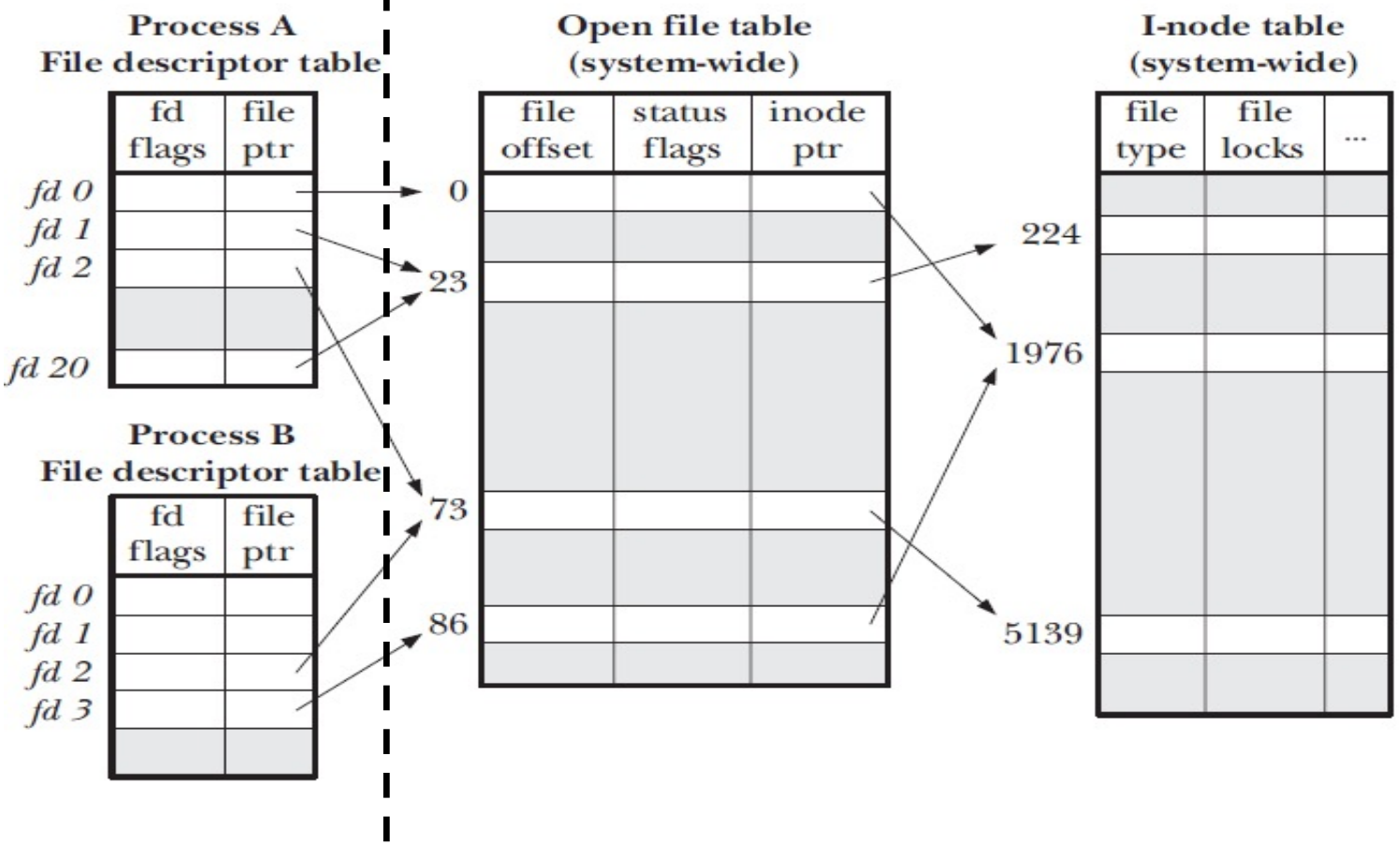
进程地址空间布局示例

- 内存资源抽象为进程地址空间



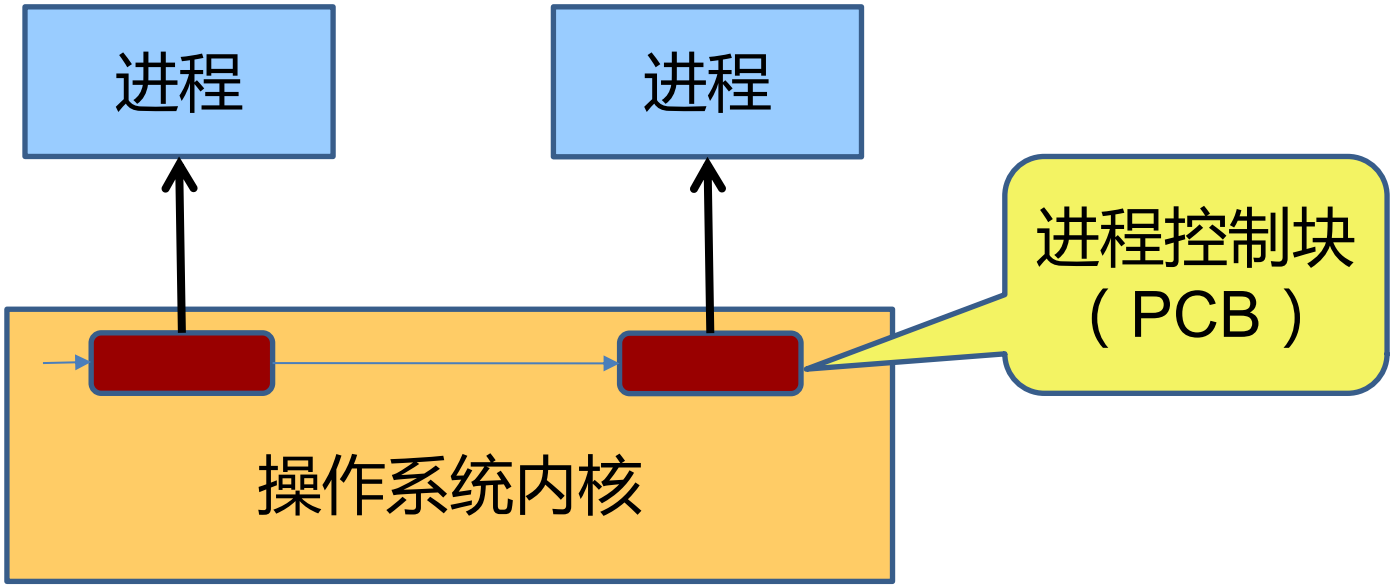
进程使用存储资源

- 存储资源抽象为文件
  - 文件描述符



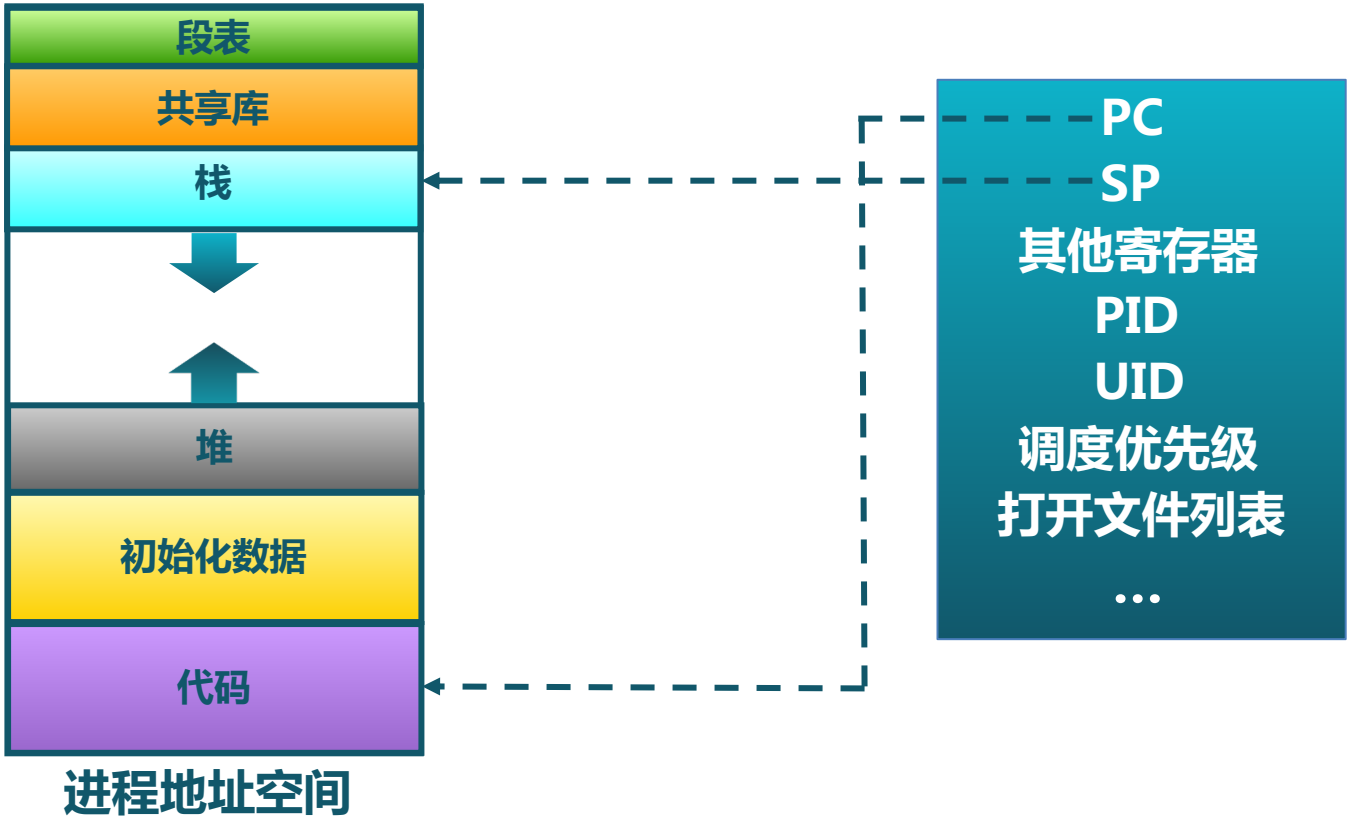
进程在内核中的表示

- 每个进程的创建与销毁都由内核负责，每个进程都需要在内核登记信息
- 内核用进程控制块（Process Control Block，PCB）来保存进程的信息
- PCB是进程在内核中的表示，也是一种便于内核管理进程的索引



进程控制块包含的信息

- 进程标识信息：进程ID、进程名称
- 与各种资源相关的信息
  - CPU、内存、存储、网络……



进程控制块包含的信息

- CPU使用信息
  - 寄存器内容
- 内存使用信息
  - 代码段、数据段、堆栈位置
  - 页表
- I/O设备使用信息
  - 文件描述符、通信端口等
- 运行状态
  - 就绪态：准备运行
  - 运行态：正在运行
  - 阻塞态：等待资源
- 优先级

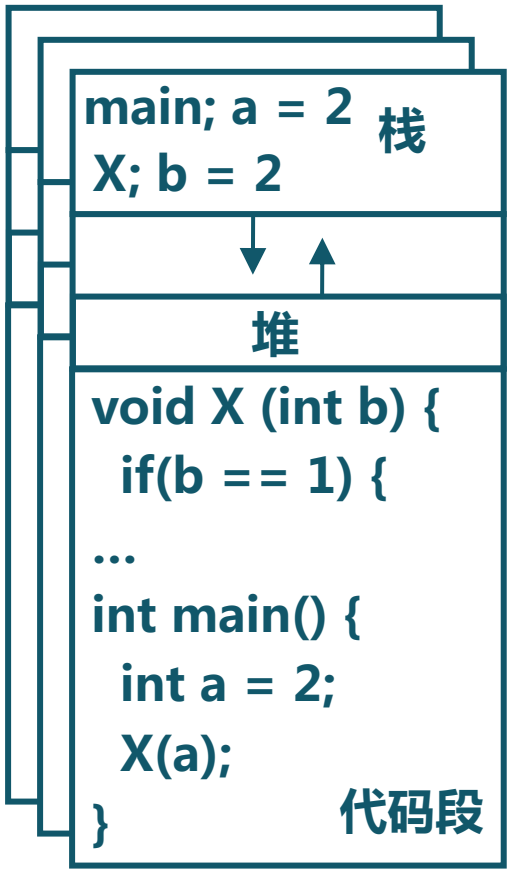
进程 vs. 程序

- 关联
  - 程序是进程的一部分（e.g. 代码段、数据段）
  - 进程是程序的动态执行
- 区别
  - 一个程序可以创建多个进程，例如一个程序同时运行多次
  - 进程还包括运行时状态和使用的资源信息

程序

```
void X (int b) {
  if(b == 1) {
...
int main() {
  int a = 2;
  X(a);
}
```

进程

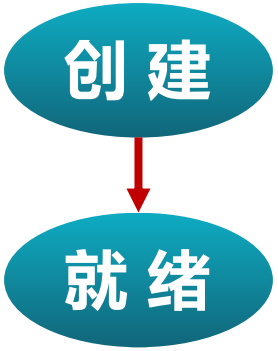


内容提要

- 进程的起源
- 进程的概念与表示
- 进程的状态
- 进程的操作原语（API）

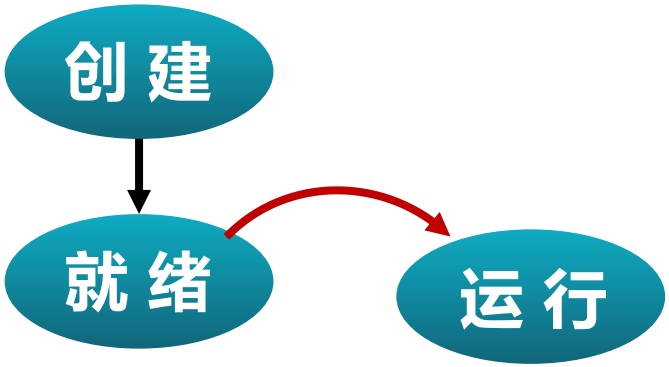
进程创建

- 什么时候会创建进程？
  - 系统初始化时（init/system进程）
  - 用户执行一个程序，创建一个新进程
    - 例如，在shell执行命令
  - 正在运行的进程执行创建进程的系统调用
    - 例如，调用fork
- 创建进程
  - 创建与初始化PCB相关数据结构
  - 创建地址空间相关数据结构，确定栈位置
  - 将数据和代码加载至内存
  - 初始化进程的状态，把进程标志为就绪态



进程运行

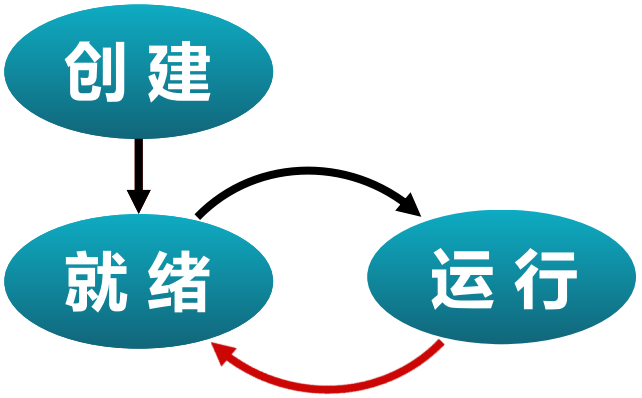
- 内核选择一个就绪的进程，为它分配一个处理器的时间片，并开始执行（时间片倒计时）



- 如何选择就绪的进程？
  - 进程调度算法
  - 进程优先级
- 思考
  - 要执行一个新进程时，之前没有运行完的进程如何处理？

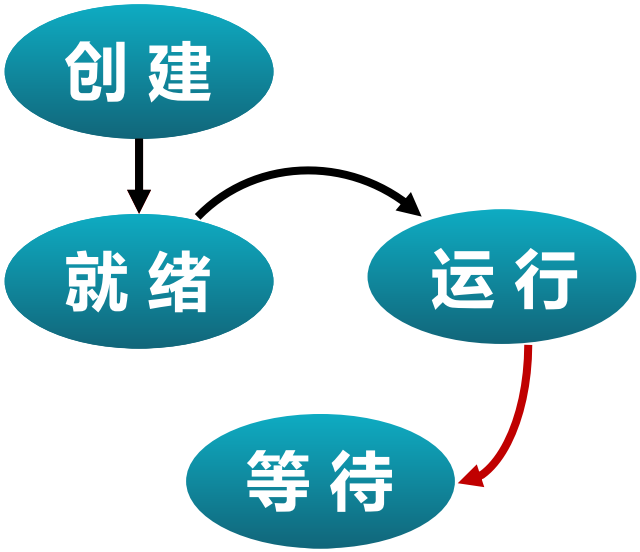
进程调度

- 当前进程时间片用完
  - 当前进程进入就绪队列
  - 按一定调度算法从就绪队列中选择一个进程执行
- 高优先级进程就绪
  - 中止当前进程执行，抢占CPU并运行（抢占式内核）
  - 低优先级进程回到就绪状态
- 非抢占式内核
  - 一个进程运行直到其主动退出，让出CPU资源；或者该进程被阻塞



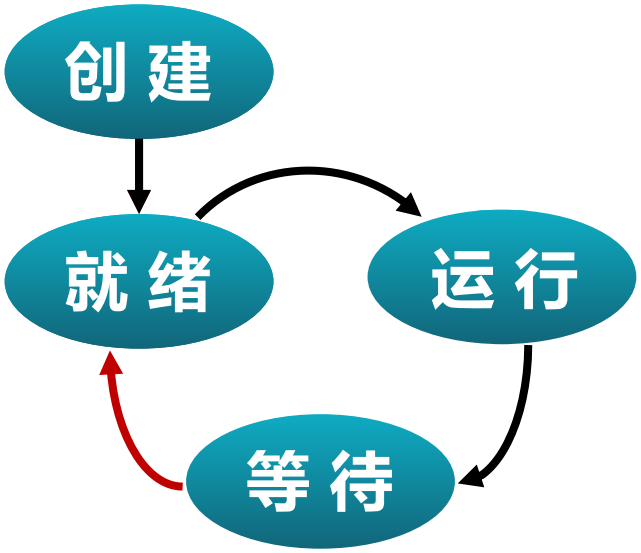
进程等待

- 进程进入等待的情况
  - 进程主动进入等待
    - 例如，sleep
  - 需要的资源没有获得
    - 例如，等待锁
  - 请求并等待系统服务，无法马上完成
    - 例如，从磁盘/网络读取数据等



进程唤醒

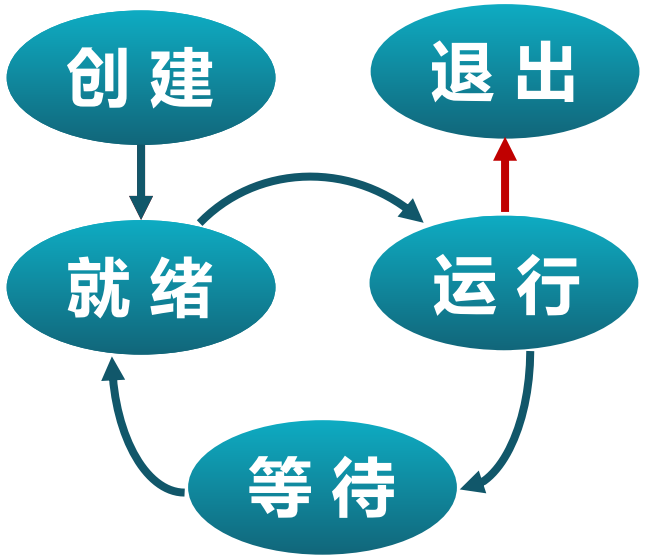
- 唤醒进程的情况
  - 被阻塞进程需要的资源被满足
  - 被阻塞进程等待的系统服务完成
- 进程只能被别的进程或操作系统唤醒



进程结束

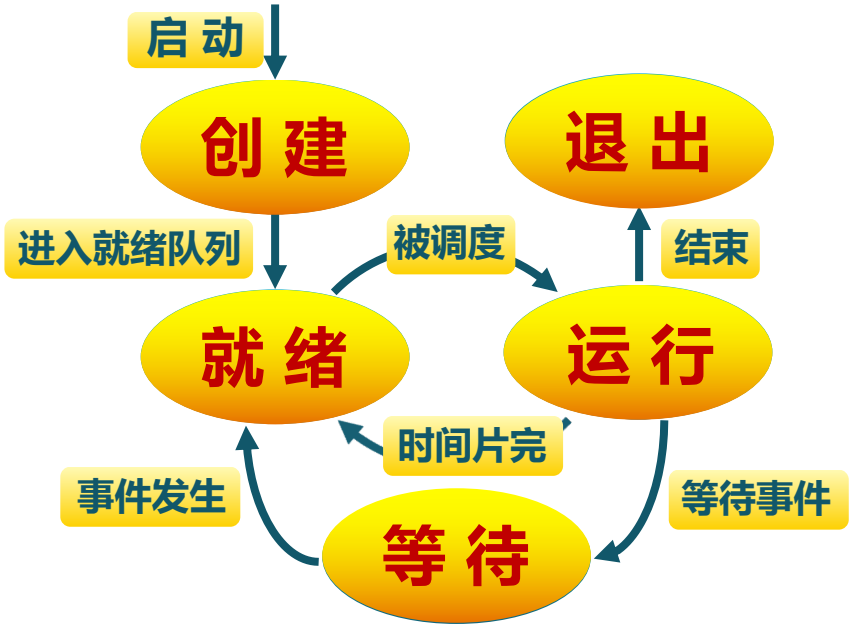
进程结束的情况

- 正常运行结束，退出
- 发生错误，退出
- 被特权用户终止



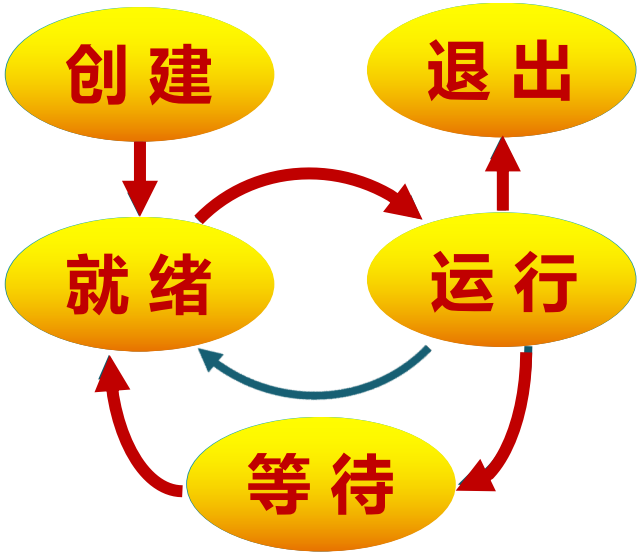
进程状态

- 进程状态转换图



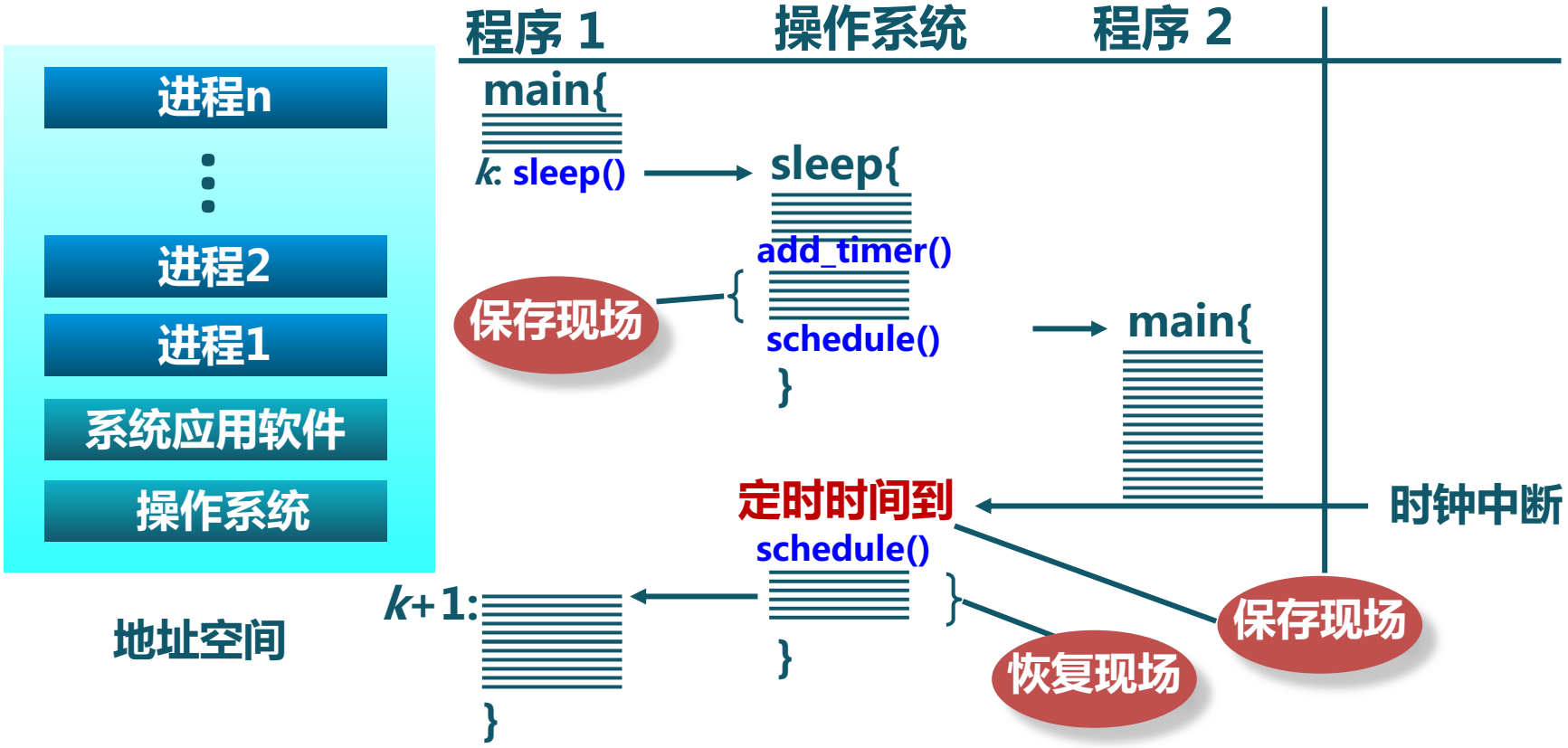
进程状态

- 示例
  - sleep()系统调用时的进程状态变化



进程切换

- 示例
  - schedule负责从就绪队列中选择进程执行



进程上下文切换

- 保存上下文
  - 保存CPU寄存器的内容（通用寄存器、状态寄存器等）
    - 例如，PC、SP、RA等寄存器
  - CPU Cache需要刷回内存么？
  - 需要将进程的内存保存到磁盘吗？
- 恢复上下文
  - 相反的操作过程
- 实现
  - 硬件自动保存/恢复：例如x86 Task Register寄存器，修改其值触发硬件保存所有寄存器值
  - 软件保存/恢复：汇编指令保存相关寄存器，例如Linux SAVE_ALL宏

内容提要

- 进程的起源
- 进程的概念与表示
- 进程的状态
- 进程的操作原语（API）

进程的操作原语（API）

- 创建和终止
  - fork， exec ，wait，kill
- 放弃CPU控制权
  - sched_yield
- 信号
  - 信号处理函数
- 同步
  - 锁，信号量，屏障

- 问题：fork, exec, wait, kill原语对PCB进行了什么操作？

fork和exec

- fork
  - 克隆出一个进程（子进程）
  - 共享当前进程（父进程）的地址空间，包括代码段，数据段、堆栈等
  - 采用写时复制机制
- exec
  - 替换掉当前进程的代码段、数据段、堆栈等

```
if ((pid = fork()) == 0) {
        /* child process */
        exec("foo");    /* does not return */
else
        /* parent */
        wait(pid);      /* wait for child to die */
```

wait和kill

- wait
  - 等待子进程结束
- kill
  - 向进程发送信号。
  - 通常用于结束进程，释放资源
  - 如何处理PCB？

总结

- 进程的起源
- 进程的概念与表示
  - CPU、内存和IO等资源的抽象
  - PCB
- 进程的状态
  - 就绪不等于运行
  - 等待和唤醒
  - 进程切换
- 进程的操作原语（API）
  - fork，exec，wait，kill