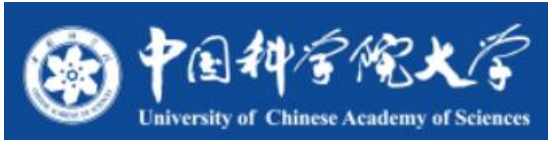
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

进程、线程与纤程

郑晨

改编声明

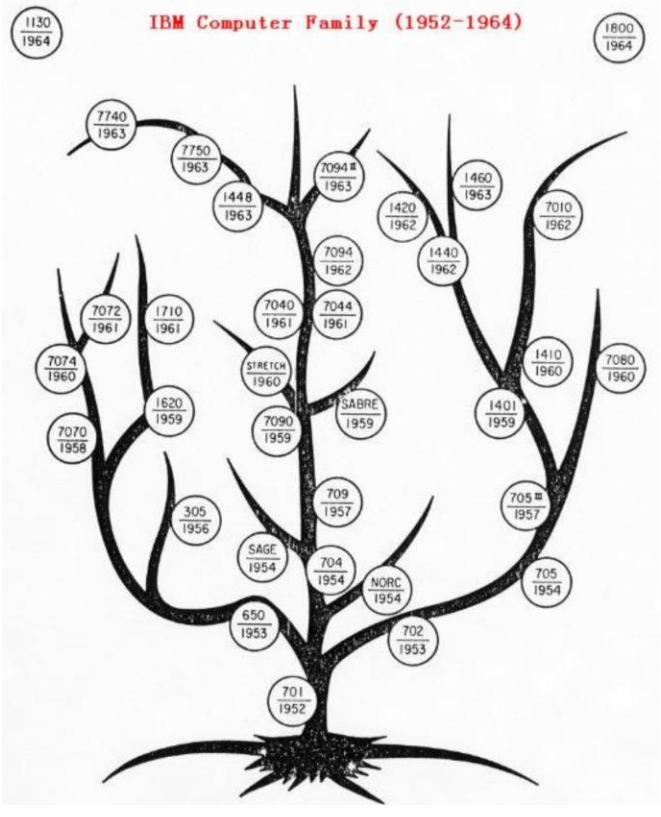
- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



进程机制的引入

进程的诞生和概念 – 进程的状态 – 数据结构 – 基本操作

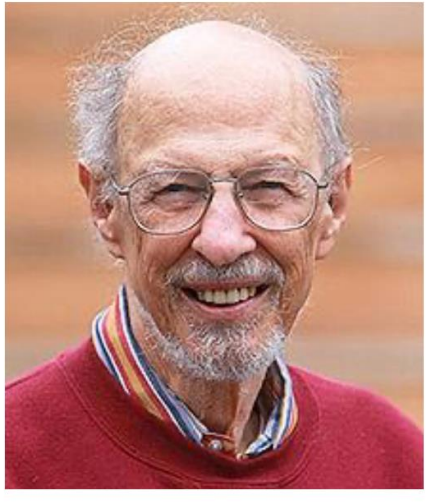
早期的IBM计算机：批处理作业



- 常驻监督程序：用来自动装填纸卡
- 客户为IBM机器开发操作系统
  - 通用汽车、通用电气/MIT
- 一次处理一个程序
  - 调试时间以天为单位

IBM 7090/7094:分时共享

- CompatibleTimeSharingSystem(CTSS)，**在IBM 7090机器上第一次实现了多个程序同时运行，“进程”开始登上历史舞台**
- FernandoJ.Corbató
- **1990年图灵奖**



从进程到线程的发展历史（一）

- **早期的操作系统（20世纪60年代）只有进程的概念**
  - 客观原因：硬件资源有限（没有多CPU、多核硬件）

- **IBM OS/360操作系统中较早出现了类似线程的概念（1967）**
  - 称为“任务”（task）

- **一般认为thread概念的提出者是**Victor N. Vyssotsky
  - Multics操作系统项目的技术领头人

从进程到线程的发展历史（二）

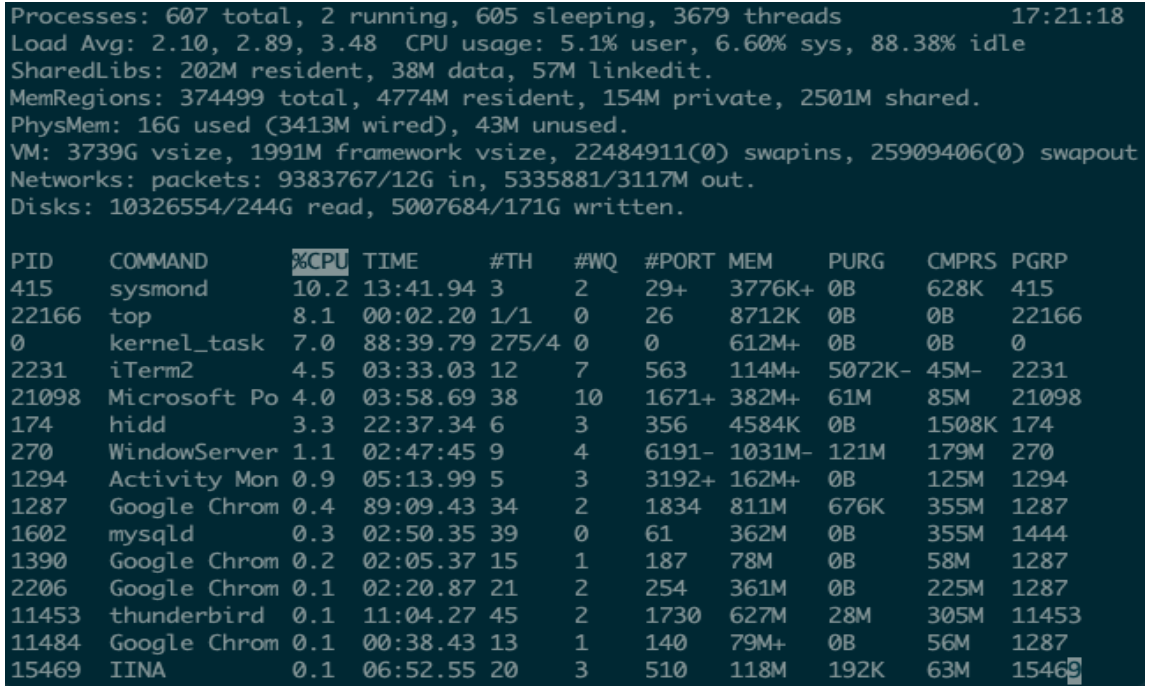
- **Linux内核（1991）最初并未提供线程支持**
- **POSIX随后（1995）定义了较为统一的线程接口**
  - **POSIX threads**，常用的*pthreads*就是一种实现
- **Red Hat的研究人员对Linux内核进行了修改**
  - 使其为线程提供原生支持，基于process实现了thread
    - 又称为“lightweight process”，即共享虚拟地址空间的进程
  - 在Linux 2.6以后进入主线（2003）

多道编程的挑战

- **需求**
  - virtual machine abstraction
    - 每个程序都期望独占机器资源
  - concurrency
    - 通过程序间并发提高硬件资源的利用
  - protection
    - 实现对共享资源的仲裁和独占资源的保护
  - Coordination
    - 程序间需要能够彼此通信合作
- **通过进程实现**
  - 多道并发
  - 隔离保护
  - 资源共享
  - 调度协作

进程定义

- **什么是进程**
  - 程序（Program）执行的一个实例（instance）



用户，程序，进程

- **用户在系统中有账户**
- **用户加载程序**
  - 多个用户可以加载相同程序吗?
  - 一个用户可以加载相同程序的多个实例吗?

- **进程是程序（Program）执行的一个实例（instance）**
  - 两个实例之间的关系是?

程序实例

```
int myval;

int main(int argc, char *argv[])
{
    myval = atoi(argv[1]);
    printf("myval is %d, loc 0x%lx\n",
                    myval, (long) &myval);
}
```

- 静态变量的地址总是相同的，但其值却是不同的

- 结论：地址不是绝对的
  - 每个进程有自己独立的内存地址空间
- 优势：
  - Compiler/linker/loader不需要关心具体地址
  - 地址空间可以大于实际内存

进程 VS 程序

- **程序不是进程**
  - 程序是静态的
    - 代码+数据
  - 进程是动态的
    - instruction execution + data + OS resource + more

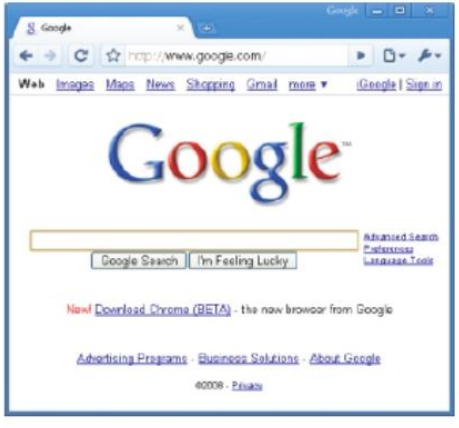
- **程序vs进程: 不是1对1的映射关系**
  - 进程不只有代码与数据
  - 一个程序可能有多个同时执行的进程
  - 一个进程可以运行多个程序

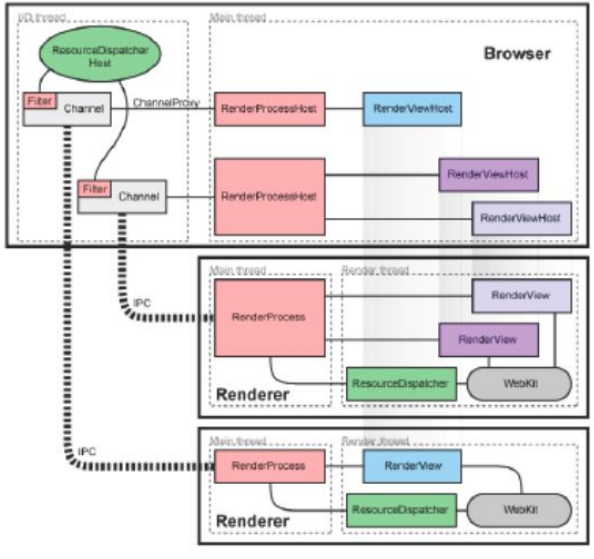
通过进程实现的两种虚拟机制

- **CPU虚拟**
  - 通过上下文(Context)环境实现对CPU的虚拟
- **内存虚拟**
  - 通过地址空间(Address)机制实现对对内存系统的虚拟
- **两种虚拟机制共同构筑了virtual machine abstraction**
  - 多道并发、隔离保护、资源共享、调度协作

Example: Thee Chrome Browser

- 多个进程：各个插件、标签页
- 如果一个页面崩溃，不会使得整个浏览器崩溃





Source: http://www.chromium.org/developers/design-documents/multi-process-architecture

进程的内涵与外延

- **进程是一种操作系统对于执行的抽象**
  - 是一种执行的实体
  - 是系统调度的单位
  - 是一个程序的动态执行上下文
- **程序执行的实体**
  - Register: PC, SP、X1-X31
  - Memory: code, data, stack, heap
- **资源分配的载体**
  - 拥有独立的地址空间:memory (address space), file descriptors, file system context, ...
  - 拥有独立I/O 状态:file descriptor table, network sockets
- **程序协调的机制**
  - 进程间通信IPC、pipe、socket

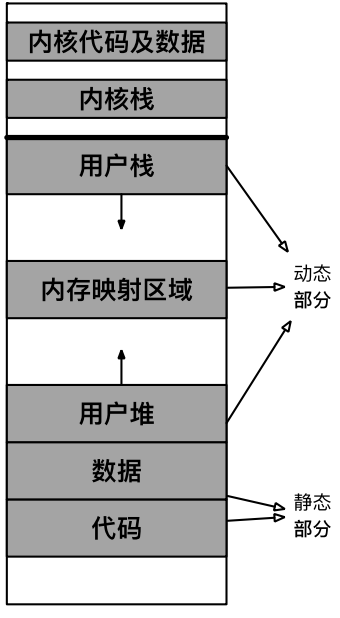
进程

进程的诞生和概念 – 进程的状态 – 数据结构 – 基本操作

进程：运行中的程序

- **进程是计算机程序运行时的抽象**
  - 静态部分：程序运行需要的代码和数据
  - 动态部分：程序运行期间的状态（程序计数器、堆、栈……）

- **进程具有独立的虚拟地址空间**
  - 每个进程都具有“独占全部内存”的假象
  - 内核中同样包含内核栈和内核代码、数据



如何表示进程：进程控制块（PCB）

- **每个进程都对应一个元数据，称为“进程控制块”PCB**
  - 进程控制块存储在内核态（为什么？）

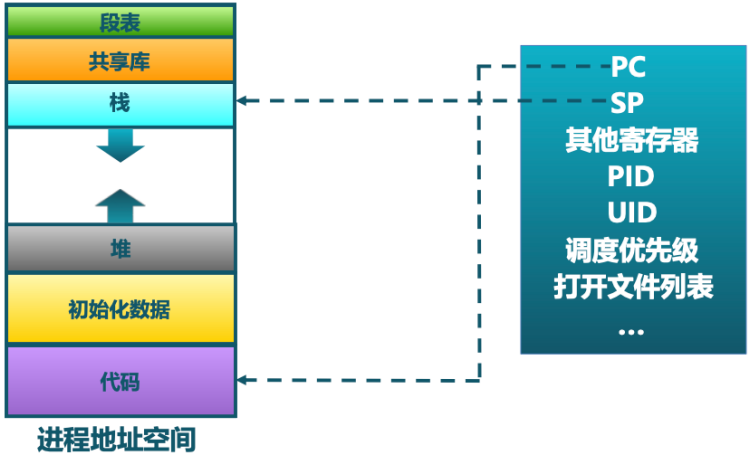
- **想一想：进程控制块里至少应该保存哪些信息？**
  - 独立的虚拟地址空间
  - 独立的执行上下文

进程的组成

- **一个进程（PCB）包括程序执行过程中的所有状态**
  - 内存地址空间
  - 执行程序的代码、数据
  - 执行stack：包含所有调用的状态
  - Program counter （PC）：指向下一个指令
  - 通用寄存器值
  - 操作系统资源集合：打开的文件、网络链接等

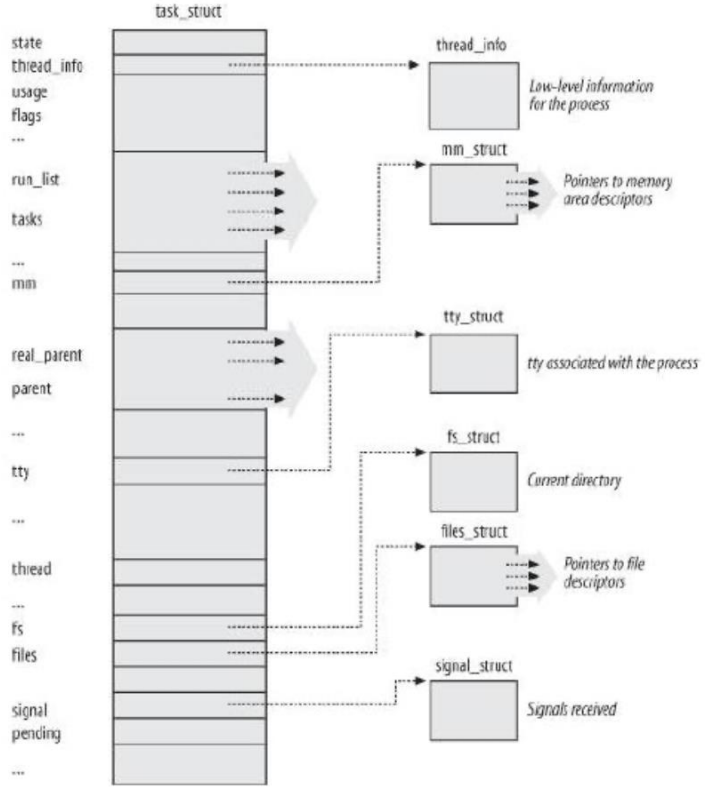
PCB

- 记录进程自身状态，包括
  - 进程状态(new, ready,running, waiting,...)
  - 机器状态(e.g., CPU register, TLB)
  - 调度与统计信息
  - 内存管理信息
  - I/O状态信息



- **将CPU硬件状态从一个进程切换到另一个进程——进程上下文切换（Context Switch）**
  - 一秒钟可以发生100到1000次
- **调度时：进程放弃CPU给其他进程**
  - 同一时刻只有一个进程在运行
  - 给重要进程更多的运行时间

PCB in Linux： task_struct



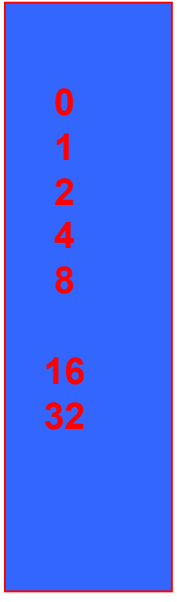
- Linux 5.5.10中670LOC

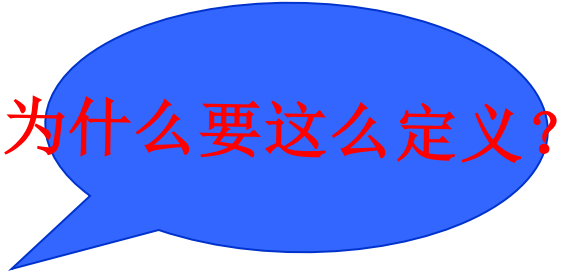
task_struct剖析（1）

```
struct task_struct {
        volatile long state;        /* -1 unrunnable, 0 runnable, >0 stopped */
        //...
        long exit_state;
        //...
}
```

进程的状态宏定义：

```
#define  TASK_RUNNING                0
#define  TASK_INTERRUPTIBLE          1
#define  TASK_UNINTERRUPTIBLE        2
#define  __TASK_STOPPED              4
#define  __TASK_TRACED               8
 /* in tsk->exit_state */
#define  EXIT_ZOMBIE                 16
#define  EXIT_DEAD                   32
//...
```





task_struct剖析（2）

```
struct task_struct {
    //...
    struct list_head tasks;        //将系统中所有进程通过双向链表链接起来！
    //...
}
```

怎样访问所有的进程呢？

```
#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )

#define next_task(p) \
        list_entry_rcu((p)->tasks.next, struct task_struct, tasks)
```

task_struct剖析（3）

```
struct task_struct {
        //...
        pid_t pid;                                          //进程标识符（线程）
        pid_t tgid;                                         //线程组的领头线程ID
        struct task_struct *group_leader;                   // threadgroup leader
        //...
}
```

系统调用 getpid() 返回什么？

Linux系统允许用户使用一个叫做进程标识符的PID来标识进程，PID顺序编号，新创建进程是前一个进程的PID加1，不过PID值有一个上限，达到上限之后再开始循环使用闲置的小PID。

task_struct剖析（4）

```
struct task_struct {
    //...
    struct task_struct __rcu *real_parent;     /* real parent process */
    struct task_struct __rcu *parent;   /* recipient of SIGCHLD, wait4() reports */

    struct list_head children;          /* list of my children */
    struct list_head sibling;           /* linkage in my parent's children list */
    //...
}
```

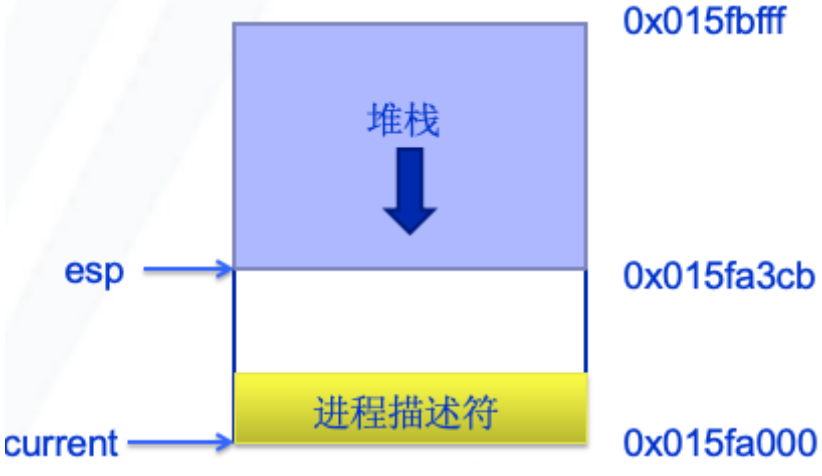
进程之间的关系：

父子关系

兄弟关系

current宏定义解析

- current宏：一个全局指针，指向当前进程的struct task_struct结构体，即表示当前进程。
- 例如current->pid就能得到当前进程的pid，current->comm就能得到当前进程的名称。



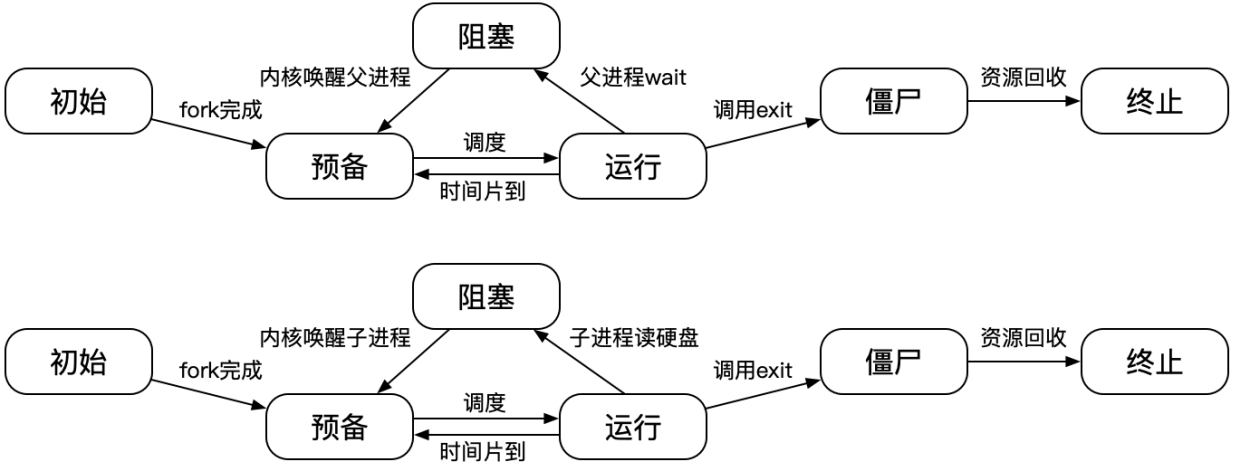
简化模型：单一线程的进程

- **假设每个进程都只有一个线程**
  - 历史上确实如此！ （如早期的UNIX操作系统）

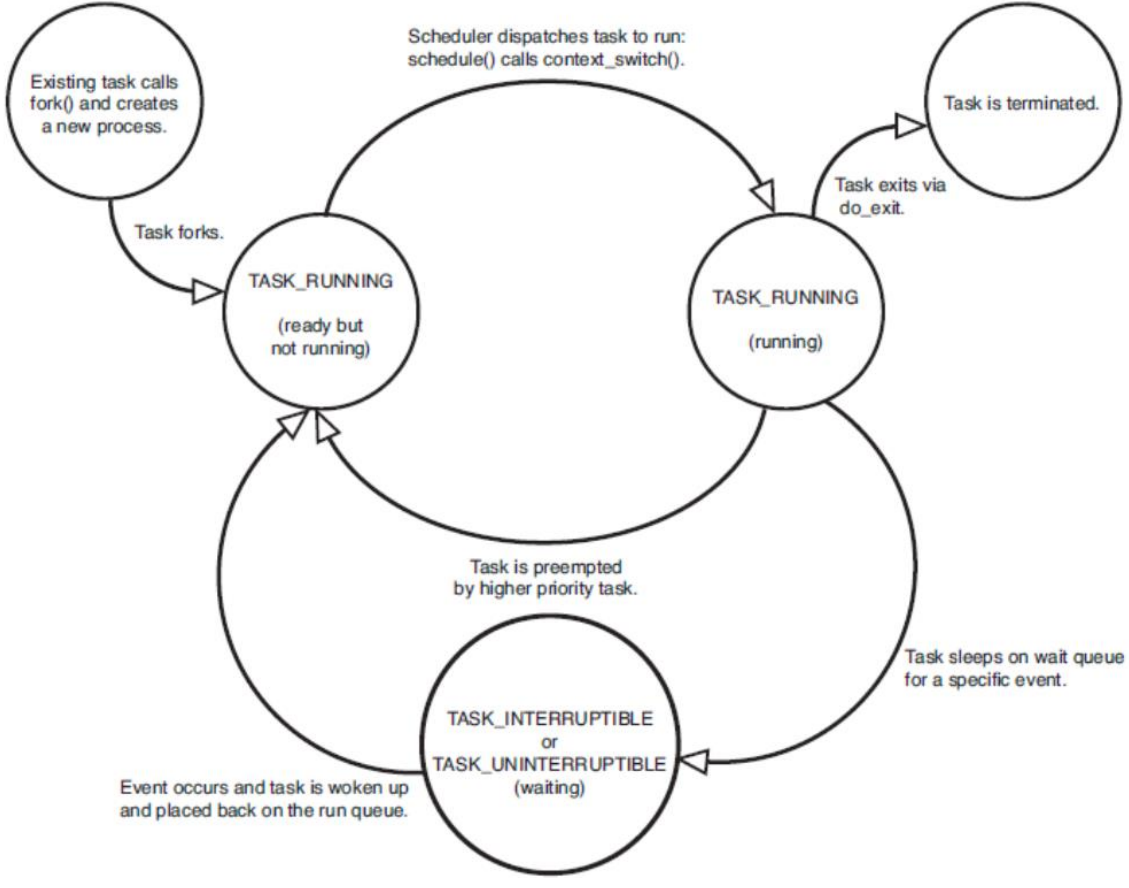
- **好处：便于理解操作系统中的各种概念**
  - 多线程使操作系统的管理更加复杂
  - 在单一线程的进程中：线程管理 / 调度 ≈ 进程管理 / 调度
  - 线程的内容将在后面介绍

进程管理：即管理进程的生命周期

- **进程自创建到终止可经历多个过程**
  - 称为进程**状态**
- **不同的系统调用和事件会影响进程的状态**

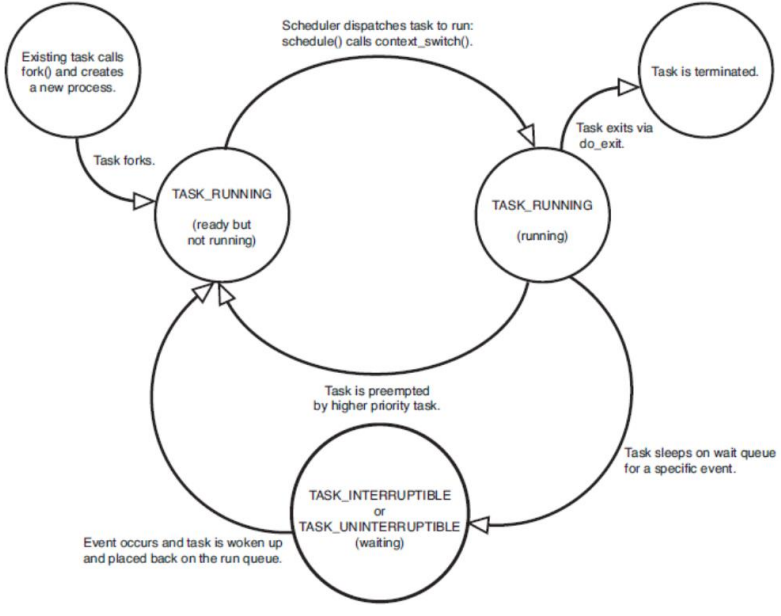


Linux进程5种状态变换流图

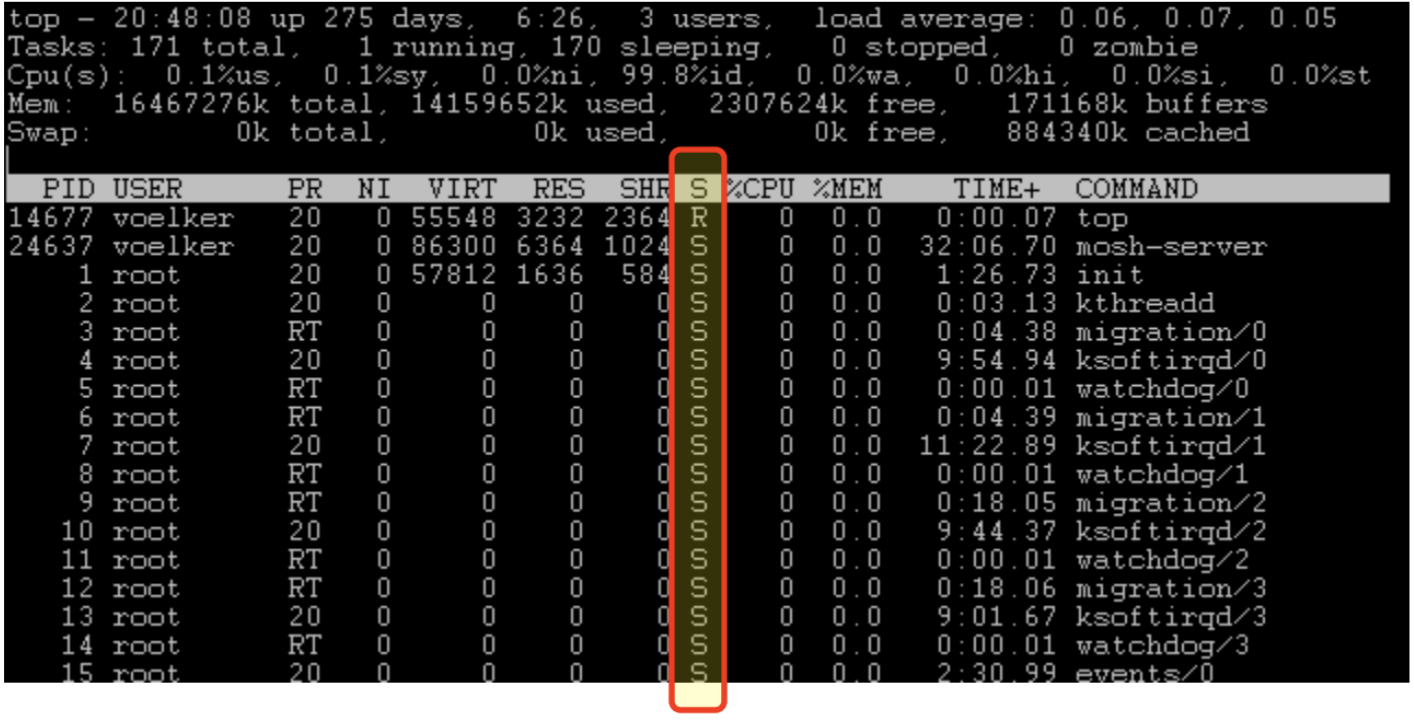


Linux进程状态

- **TASK_RUNNING(R)**
  - 可运行，但不等于正在运行，可能在运行队列中
- **TASK_INTERRUPTIBLE(S)**
  - 可被中断的睡眠状态，可被信号唤醒
- **TASK_UNINTERRUPTIBLE(D)**
  - 不可被中断的睡眠状态，俗称“D住了”，不能被信号唤醒
- **__TASK_TRACED**
  - 被跟踪状态，调试用
- **__TASK_STOPPED**
  - 停止执行状态，通常在于收到了SIGSTOP等状态



Linux进程状态



```
top - 20:48:08 up 275 days,  6:26,  3 users,  load average: 0.06, 0.07, 0.05
Tasks: 171 total,   1 running, 170 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.1%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  16467276k total, 14159652k used,  2307624k free,   171168k buffers
Swap:        0k total,        0k used,        0k free,   884340k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14677 voelker   20   0 55548 3232 2364 R    0  0.0   0:00.07 top
24637 voelker   20   0 86300 6364 1024 S    0  0.0  32:06.70 mosh-server
    1 root      20   0 57812 1636  584 S    0  0.0   1:26.73 init
    2 root      20   0     0    0    0 S    0  0.0   0:03.13 kthreadd
    3 root      RT   0     0    0    0 S    0  0.0   0:04.38 migration/0
    4 root      20   0     0    0    0 S    0  0.0   9:54.94 ksoftirqd/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.01 watchdog/0
    6 root      RT   0     0    0    0 S    0  0.0   0:04.39 migration/1
    7 root      20   0     0    0    0 S    0  0.0  11:22.89 ksoftirqd/1
    8 root      RT   0     0    0    0 S    0  0.0   0:00.01 watchdog/1
    9 root      RT   0     0    0    0 S    0  0.0   0:18.05 migration/2
   10 root      20   0     0    0    0 S    0  0.0   9:44.37 ksoftirqd/2
   11 root      RT   0     0    0    0 S    0  0.0   0:00.01 watchdog/2
   12 root      RT   0     0    0    0 S    0  0.0   0:18.06 migration/3
   13 root      20   0     0    0    0 S    0  0.0   9:01.67 ksoftirqd/3
   14 root      RT   0     0    0    0 S    0  0.0   0:00.01 watchdog/3
   15 root      20   0     0    0    0 S    0  0.0   2:30.99 events/0
```

进程创建：fork()

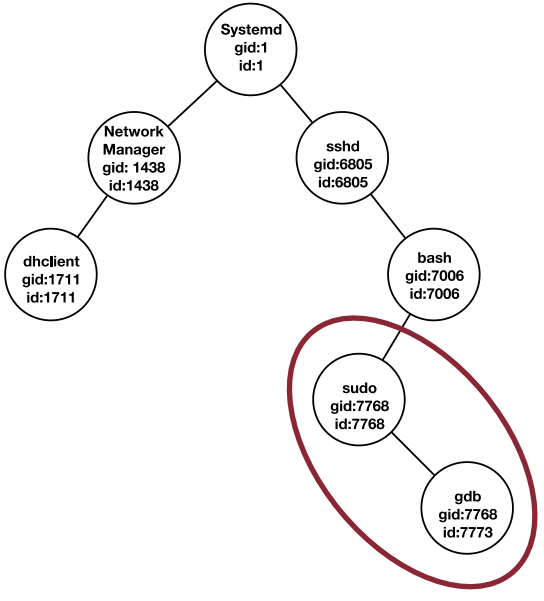
- **语义：为调用进程创建一个一模一样的新进程**
  - 调用进程为**父进程**，新进程为**子进程**
  - 接口简单，无需任何参数

- **fork后的两个进程均为独立进程**
  - 拥有不同的进程id
  - 可以并行执行，互不干扰（除非使用特定的接口）
  - 父进程和子进程会共享部分数据结构（内存、文件等）

进程树与进程组

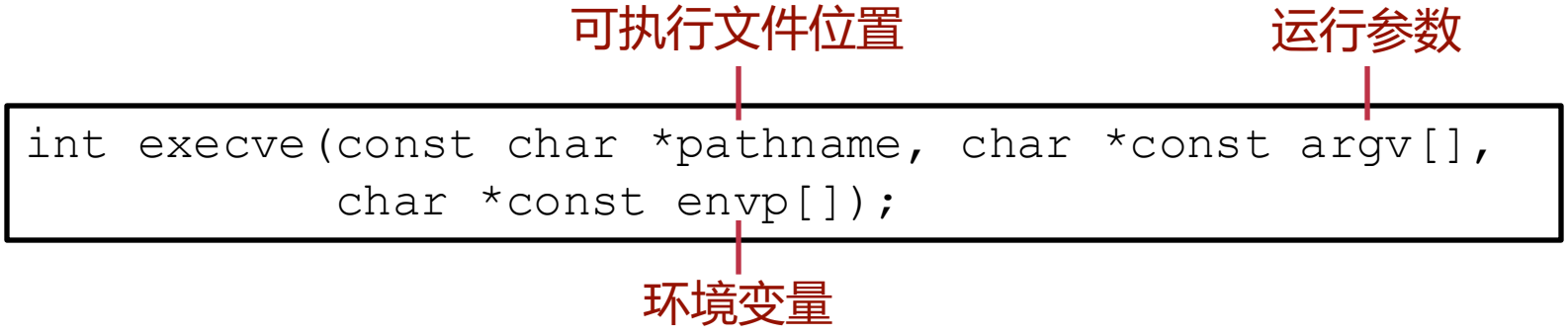
- **fork为进程之间建立了父进程和子进程的关系**
  - 进程之间建立了树型结构
  - Linux可使用ps命令查看

- **多个进程可以属于同一个进程组**
  - 子进程默认与父进程属于同一个进程组
  - 可以向同一进程组中的所有进程发送信号
  - 主要用于shell程序中



进程的执行：exec

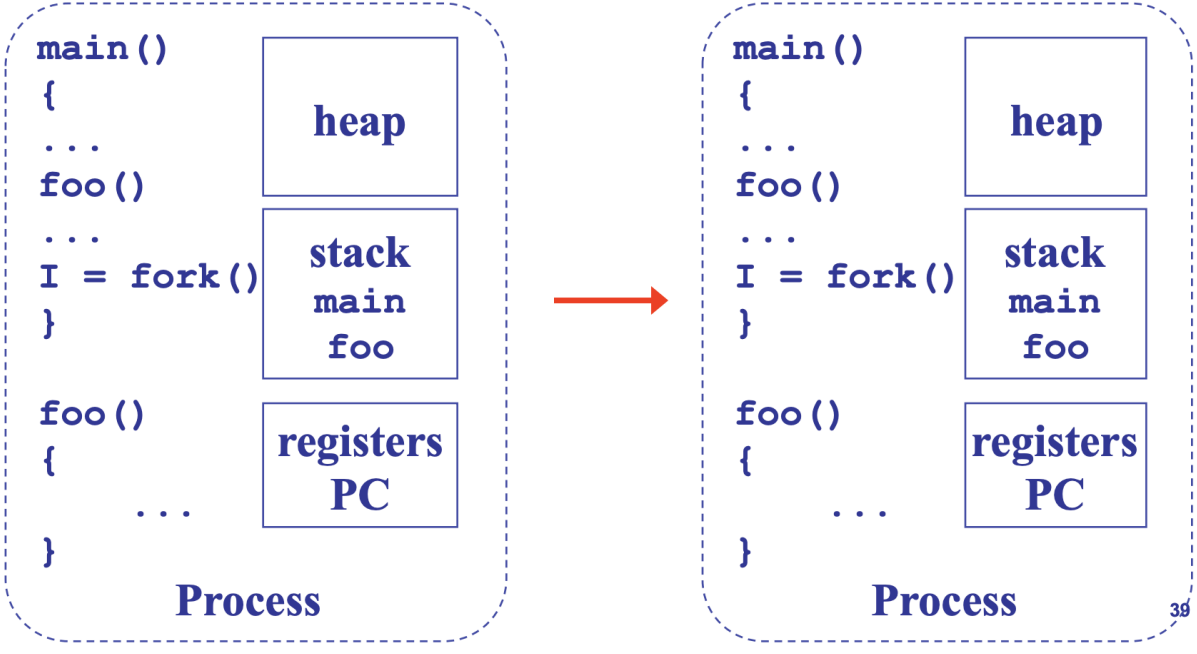
- **为进程指定可执行文件和参数**



- **在fork之后调用**
  - exec在载入可执行文件后会重置地址空间

进程API

- Fork() system call 创建当前进程的副本



Fork系统调用

- **Fork()**
  - 创建和初始化一个新的PCB
  - 创建一个新的地址空间（但，复制父进程内容）
  - 初始化地址空间：复制父进程的整个地址空间内容
  - 初始化内核资源（指向父进程使用的资源，如open files）
  - 将PCB放于ready队列

- **Fork return twice**
  - 将子进程的pid返回父进程，返回“0”给子进程

Fork()

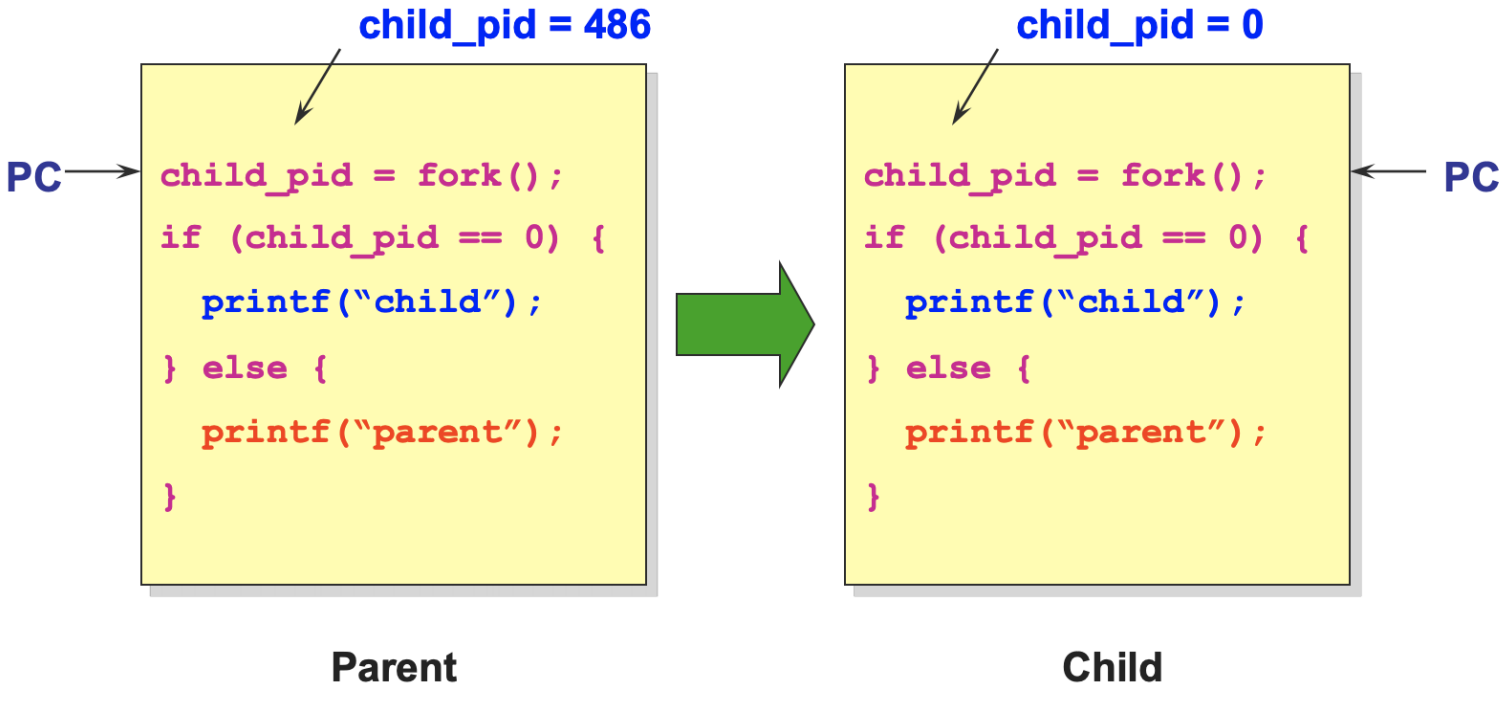
```
int main(int argc, char *argv[])
{
   char *name = argv[0];
   int child_pid = fork();
   if (child_pid == 0) {
        printf("Child of %s is %d\n", name, getpid());
        return 0;
   } else {
        printf("My child is %d\n", child_pid);
        return 0;
   }
}
```

- 这段代码会输出什么?

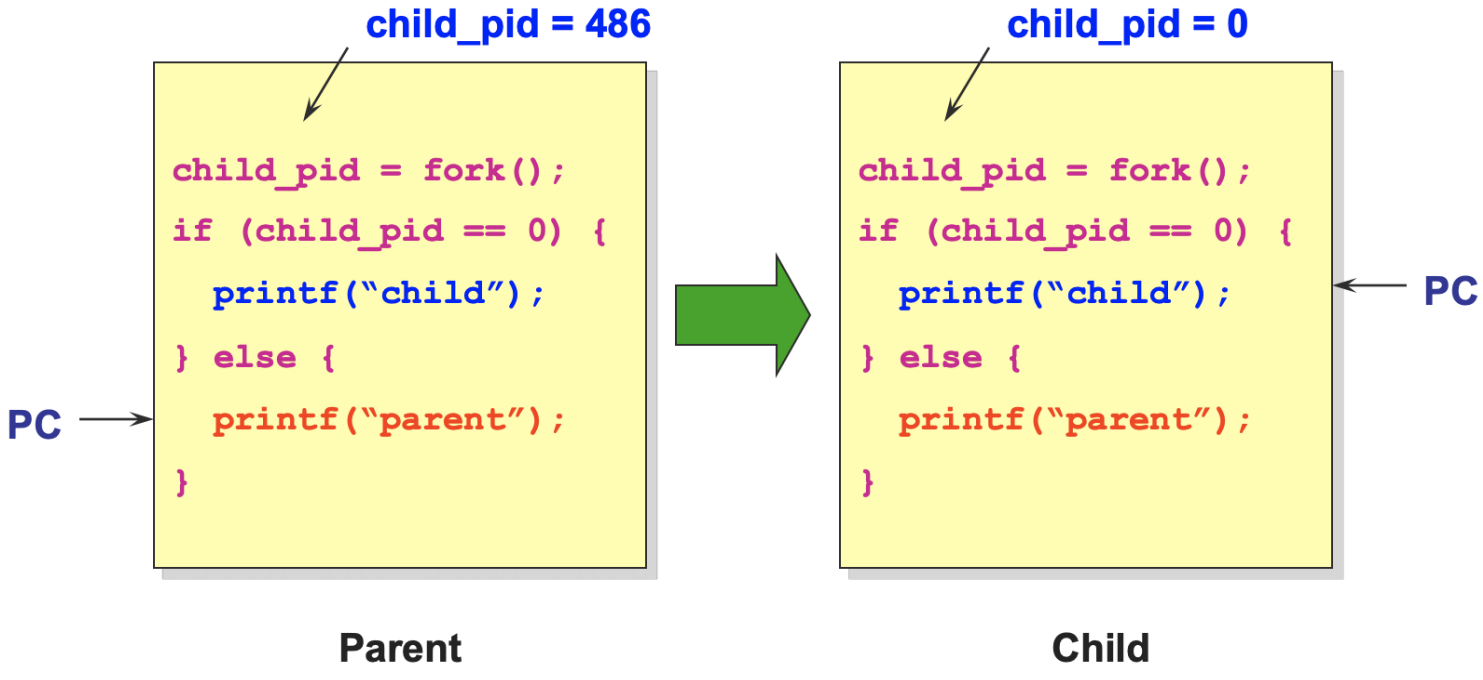
Fork()

```
alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
```

复制地址空间


child_pid = 486
PC
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
Parent
child_pid = 0
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Child

差异


child_pid = 486
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Parent
child_pid = 0
child_pid = fork();
if (child_pid == 0) {
printf("child");
} else {
printf("parent");
}
PC
Child

Fork()

```
alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> a.out
Child of a.out is 498
My child is 498
```

- **为什么顺序不同?**

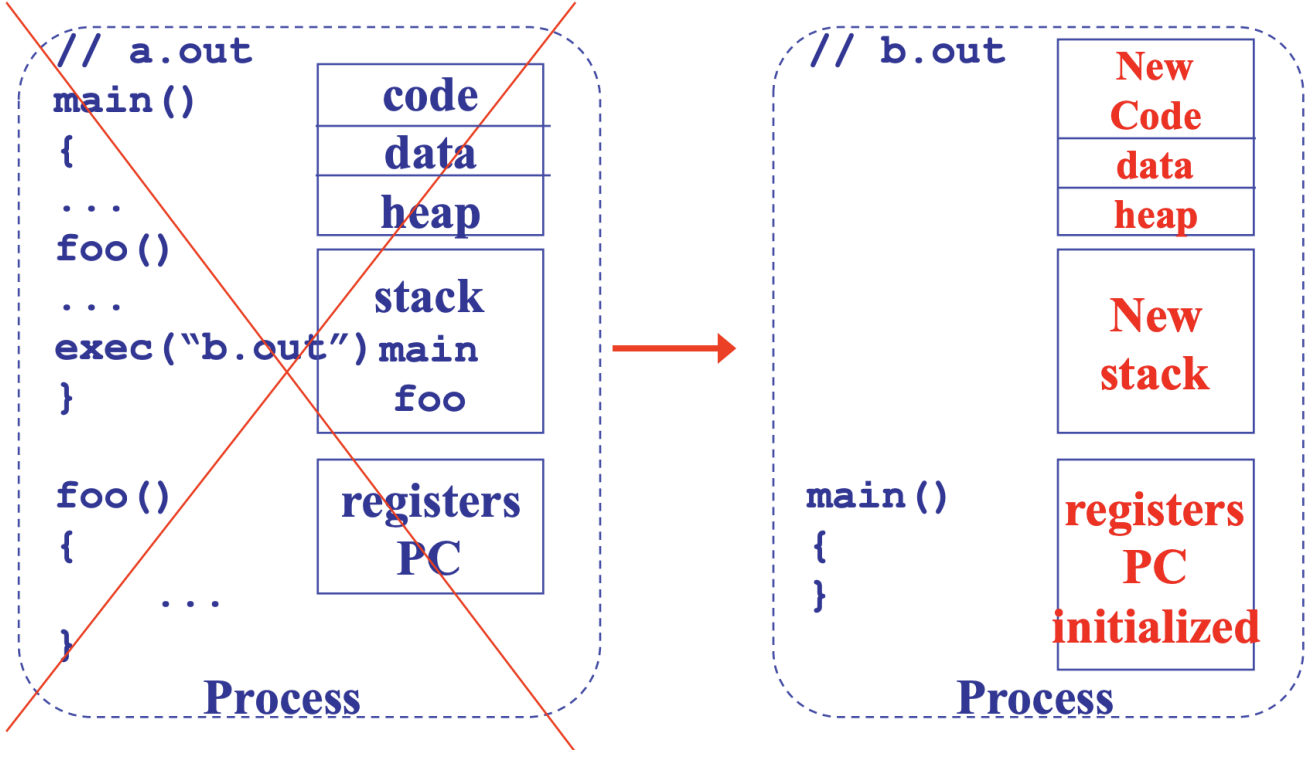
Why fork()

- 父子进程协同工作
- 依赖父进程的数据完成任务

- 例：

```
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
      Handle client request and exit
    } else {
      Close socket
    }
}
```

Exec()



- 执行exec()后，进程的什么没有改变?

写时拷贝（Copy-On-Write）

- **早期的fork实现：将父进程直接拷贝一份**
  - 性能差：时间随占用内存增加而增加
  - 无用功：fork之后如果调用exec，拷贝的内存就作废了

- **基本思路：只拷贝内存映射，不拷贝实际内存**
  - 性能较好：一条映射至少对应一个4K的页面
  - 调用exec的情况里，减少了无用的拷贝

fork的优缺点分析

- **fork的优点**
  - 接口非常简洁
  - 将进程“创建”和“执行”（exec）解耦，提高了灵活度
  - 刻画了进程之间的内在关系（进程树、进程组）
- **fork的缺点**
  - 完全拷贝过于粗暴（不如clone）
  - 性能差、可扩展性差（不如vfork和spawn）
  - 不可组合性 (例如：fork() + pthread())

fork的替代接口

- **vfork：类似于fork，但让父子进程共享同一地址空间**
  - 优点：连映射都不需要拷贝，性能更好
  - 缺点：
    - 只能用在”fork + exec”的场景中
    - 共享地址空间存在安全问题
- **轶事：vfork的提出最初就是为了解决fork的性能问题**
  - 但写时拷贝拯救了fork

> Since this function is hard to use correctly from application software, it is recommended to use posix_spawn(3) or fork(2) instead.

fork的替代接口

- **posix_spawn: 相当于fork + exec**
  - 优点：可扩展性、性能较好
  - 缺点：不如fork灵活

- **clone: fork的“进阶版”，可以选择性地不拷贝内存**
  - 优点：高度可控，可依照需求调整
  - 缺点：接口比fork复杂，选择性拷贝容易出错

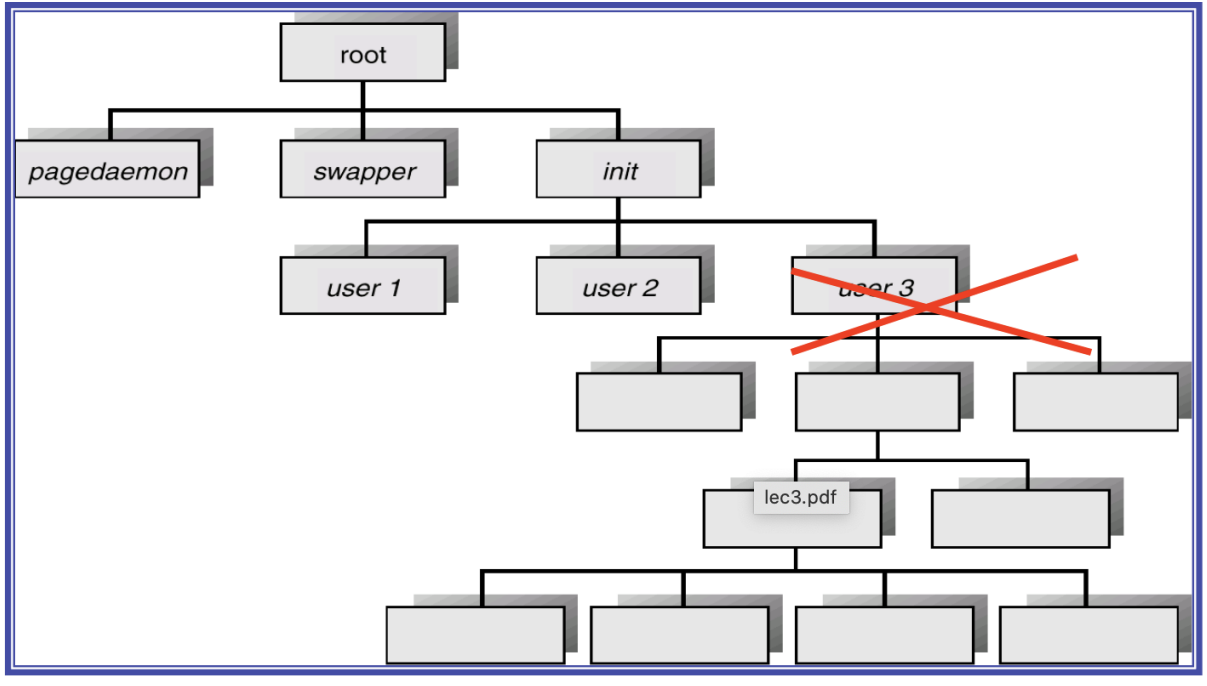
Exit()

- **Exit()系统调用**
  - 结束进程执行
  - 主要是为了释放资源

- **Exit()执行**
  - 终结所有线程（下一节）
  - 关闭所有文件描述符、网络链接
  - 释放占用的内存 （并将内存页换出到磁盘）
  - 删除PCB

- **为什么需要OS作这些操作?**

Linux系统进程树



- **如果一个父进程先于子进程消失会发生什么？**
  - Orphaned children

Orphaned children in Linux

- **Orphaned children process will set process #1 as parent**
- **From Linux 3.4**
  - 进程可以触发prctl()系统调用，with PR_SET_CHILD_SUBREAPER
  - Orphaned children proces将会以最近的ancestor process为父进程，不一定为process #1

线程

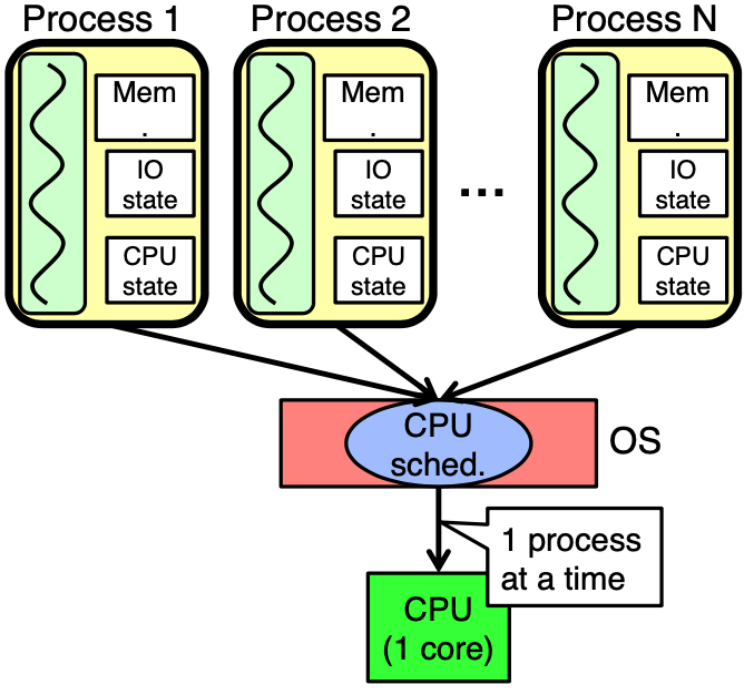
线程的概念 - 线程模型 - 相关数据结构 - 基本操作

为什么需要线程?

- **创建进程的开销较大**
  - 包括了数据、代码、堆、栈等
- **进程的隔离性过强**
  - 进程间交互：可以通过进程间通信（IPC），但开销较大
- **进程内部无法支持并行**

进程回顾

- **进程：程序执行的一个实例**
  - 地址空间、操作系统资源、统计信息、执行状态（PC、SP、regs等）
- **进程开销**
  - 创建开销
    - Task_struct in Linux
    - 所有数据需要创建和初始化
  - 通信开销
    - 进程间地址空间隔离
    - 操作系统需要介入进程间通信
    - 至少一次数据拷贝
  - 切换开销
    - CPU状态保存
    - 内存地址空间切换
  - 隔离保护：CPU、Memory、I/O

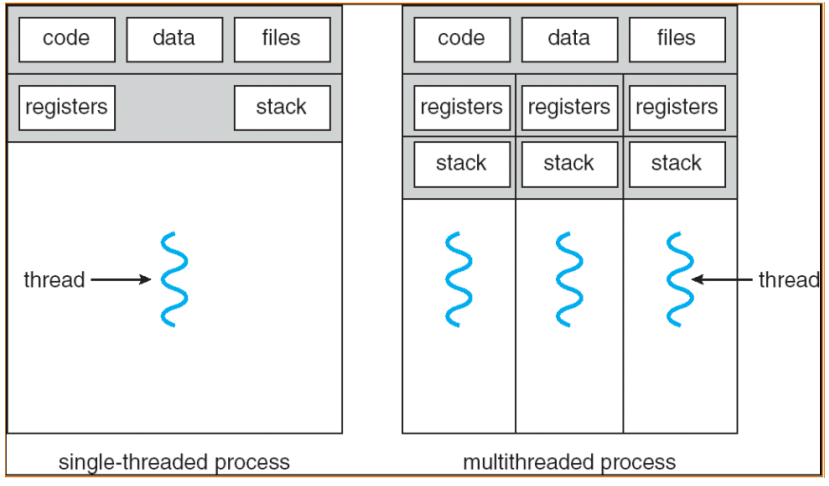


进程间通信

- **进程创建时**
  - 父进程在fork()子进程时将所有数据和状态传递给子进程

- **操作系统提供通信机制**
  - 共享内存：多个进程同时读写同一块内存区域；隐蔽信道
    - 系统调用来声明共享区域
    - 共享内存映射成功后，不需要OS介入
  - 消息传递：通过send()/receive()系统调用建立的明确通信
- **IPC**
  - 开销较大——系统调用和消息传递
  - Lightweight IPC

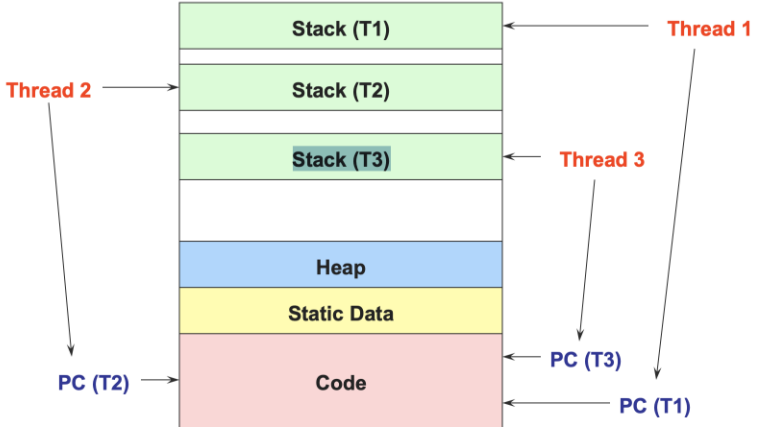
线程概念的引入



- **Key idea：分离进程和他的执行状态**
- **进程是资源分配的主体：基于进程分配地址空间、内存、I/O等**
  - 传统进程只包含一个线程
- **线程是指令执行的主体：主要基于线程管理CPU机器状态、栈执行环境等**
  - 多线程进程包含多个线程
  - 线程在同一进程的地址空间，可共享code、data、same privileges and resources (files, sockets, etc.)
- **一个线程对应到一个进程，而进程可以拥有多个线程**
- **线程是最基本的执行单元，而进程成为线程执行的容器**

Threads in a process

- 切换开销低
  - 只切换CPU寄存器
  - 地址空间和资源不变
- 创建开销低
  - 大量数据可以共享
  - Sometimes，不需要陷入内核
- 隔离保护
  - CPU:有
  - Memory/IO：无
- 共享通信开销低
  - 天生共享内存



线程讨论

- **共享部分：对在同一进程的所有线程可见**
  - 内存状态（全局变量、堆）
  - I/O状态（文件系统、网络连接状态等）
- **私有部分：线程私有的机器状态**
  - 程序执行指针（Program counter， EIP on x86）
  - 其他CPU相关寄存器register
  - 线程执行环境栈：函数调用参数、临时变量、返回PC
- **资源回收：**
  - 线程只回收栈
  - 进程回收所有资源
- **线程并发性**
  - 相较进程，线程更容易实现I/O overlapping
  - Web Server：服务大量并发网络请求

线程小结：更加轻量级的运行时抽象

- **线程只包含运行时的状态**
  - 静态部分由**进程**提供
  - 包括了执行所需的**最小**状态（主要是寄存器和栈）

- **一个进程可以包含多个线程**
  - 每个线程共享同一地址空间（方便数据共享和交互）
  - 允许进程内并行

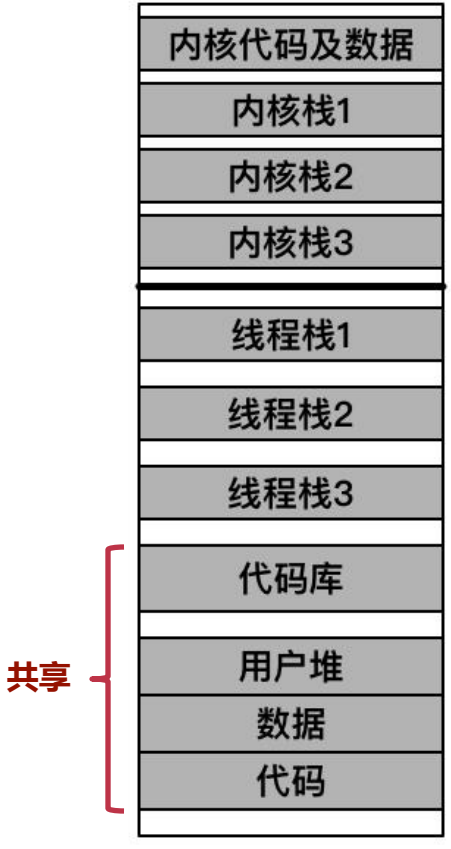
进阶模型：多线程的进程

- **一个进程可以包含多个线程**

- **一个进程的多线程可以在不同处理器上同时执行**
  - 调度的基本单元由进程变为了线程
  - 每个线程都有**状态**
  - 上下文切换的单位变为了线程

多线程进程的地址空间

- 每个线程拥有自己的栈
- 内核中也有为线程准备的内核栈
- 其它区域共享
  - 数据、代码、堆……



用户态线程与内核态线程

- **根据线程是否受内核管理，可以将线程分为两类**
  - 内核态线程：内核可见，受内核管理
  - 用户态线程：内核不可见，不受内核直接管理
- **内核态线程**
  - 由内核创建，线程相关信息存放在内核中
- **用户态线程（纤程）**
  - 在应用态创建，线程相关信息主要存放在应用数据中

线程调度机制

- **不再以进程为调度单位，而是以线程**
  - 内核怎么选择PCB?
- **两种基本方法：**
  - 内核知道进程中的所有线程，并去调度线程
  - 内核不知道线程存在，将线程事做普通进程管理调度，利用用户态的调度器来调度每个进程中的多个线程
- **两种方式的优劣?**
  - 想象不同方式下线程切换的开销
  - 谁决定线程的调度顺序
  - 当一个线程调用阻塞的系统调用呢

用户态线程的调度

- **如果线程属于同一个进程**
  - 由用户态线程库进行管理
    - 只需要对TCB信息进行load/store
  - 操作系统不需要介入

- **如果线程属于不同进程**
  - 正常的进程切换方式
    - 由OS进行（trap in/out of kernel）
    - 操作系统需要load/store PCB 和TCB info

用户态线程的调度（2）

- **非抢占式调度**
  - 没有timer机制强制让线程让出CPU
  - 线程必须自愿放弃CPU给其他线程 e.g. pthread_yield
  - 调度器不考虑线程历史
  - 执行是协作式（co-operative），而非强迫式competitive

- **抢占式调度**
  - 通过信号模拟中断，触发强制调度 e.g. alarm
  - 会带来许多实现问题
    - 如对应的信号量无法作他用

用户态线程的阻塞问题

- **线程执行read()读取磁盘发生阻塞**
  - 等待数据从磁盘读回
  - 进程阻塞直到硬盘I/O完成

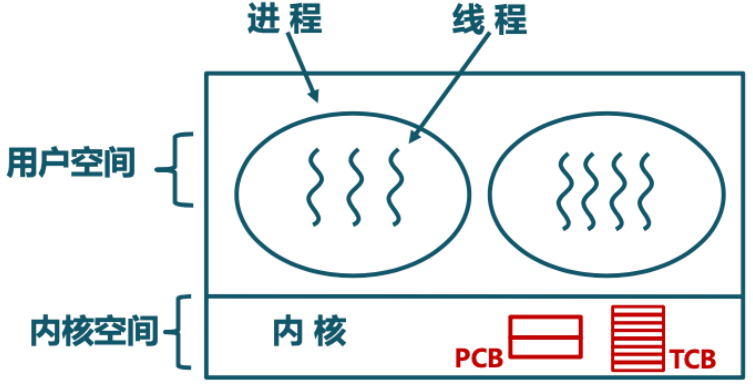
- **解决方案：wrapper functions**
  - 重载系统调用（提供同名另一个版本的系统调用）
  - 在进入内核时检查是否会发生阻塞
  - E.g. select() before read()
  - 如果调用阻塞，调度另外一个线程执行
  - 过于复杂：需要处理所有的阻塞调用
  - 好处：批处理方式执行，将同步转化为异步

用户态线程的问题

- **如果一个线程调用系统调用**
  - 阻塞的系统调用会阻塞整个进程
- **用户态线程对内核透明**
  - 不能很好的与OS进行交互和集成
- **操作系统可能会成为性能瓶颈**
  - 调度有idle线程的进程
  - 阻塞一个发起了I/O请求的进程，即使该进程的其他线程还可以继续执行
  - 当一个线程hold锁时，不调度该线程所在进程

内核态线程

- **操作系统管理所有的线程和进程**
  - 线程管理操作全部由内核完成，包括创建、销毁、调度、同步等
  - 线程是调度基本单元



- **线程调度管理**
  - PCB不再被调度
  - 如果一个线程阻塞，内核可以调度同一进程中的其他线程执行

内核态线程

- **创建和管理较慢**
  - 需要陷入内核
  - 内核维护更多的数据结构

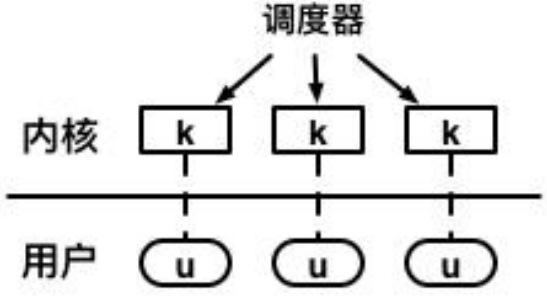
- **与内核集成较好**
  - 一个阻塞的系统调用不会阻塞整个进程

用户态和内核态线程

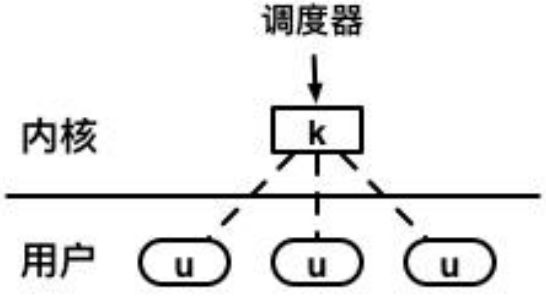
- **两种实现机制的区别?**
- **同时使用用户态和内核态线程**
  - 将一个内核态线程和一个用户态线程绑定
  - 或者将多个用户态线程在多个内核态线程上复用
- **Java virtual machine （JVM)**
  - Java线程是用户态线程
  - 在以前版本，每个进程只有一个内核态线程
    - 需要将全部java线程在这一个内核态线程上复用
  - 现在的操作系统中
    - 可以在多个内核态线程上复用
    - Java线程可以远多于内核态线程

线程模型

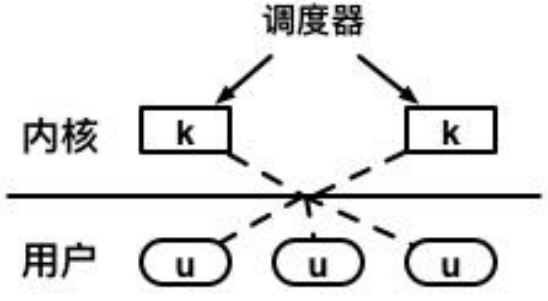
- **线程模型表示了用户态线程与内核态线程之间的联系**
  - 多对一模型：多个用户态线程对应一个内核态线程
  - 一对一模型：一个用户态线程对应一个内核态线程
  - 多对多模型：多个用户态线程对应多个内核态线程



一对一模型



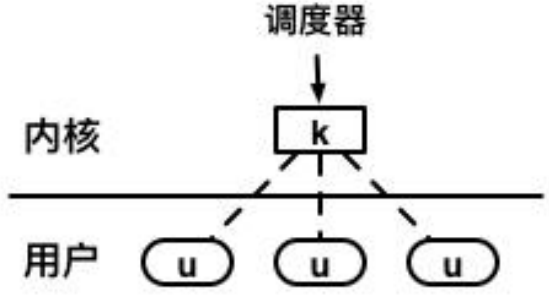
多对一模型



多对多模型

多对一模型

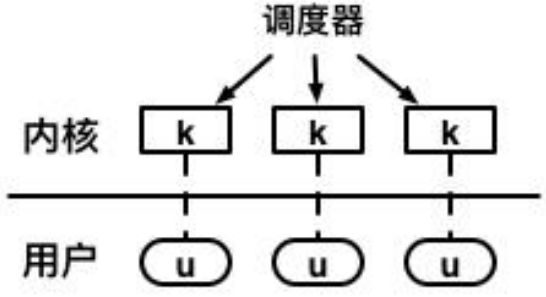
- **将多个用户态线程映射给单一的内核线程**
  - 优点：内核管理简单
  - 缺点：可扩展性差，无法适应多核机器的发展
- **在主流操作系统中被弃用**
- **用于各种用户态线程库中**



一对一模型

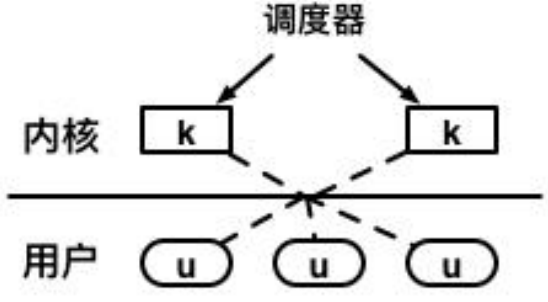
- **每个用户线程映射单独的内核线程**
  - 优点：解决了多对一模型中的可扩展性问题
  - 缺点：内核线程数量大，开销大

- **主流操作系统都采用一对一模型**
  - Windows、Linux、OS X……



多对多模型（又叫Scheduler Activation）

- **N个用户态线程映射到M个内核态线程（N > M）**
  - 优点：解决了可扩展性问题（多对一）和线程过多问题（一对一）
  - 缺点：管理更为复杂
- **Solaris在9之前使用该模型**
  - 9之后改为一对一
- **在虚拟化中得到了广泛应用**



线程池（Thread Pool）

- **问题：服务器应用优化**
  - 为每个服务请求创建线程开销大
    - 被创建线程在服务完毕后退出
  - 请求的增多-->带来更多的线程数量，更大的服务器开销

- **Solution：线程池（Thread pool）**
  - 预先创建一定数量的线程，等待服务
  - 用户请求到达时，唤醒线程 （唤醒线程开销远低于创建开销）
  - 请求结束时，线程并不直接退出，而是释放回线程池
  - 确定线程池的容量

线程的相关数据结构：TCB

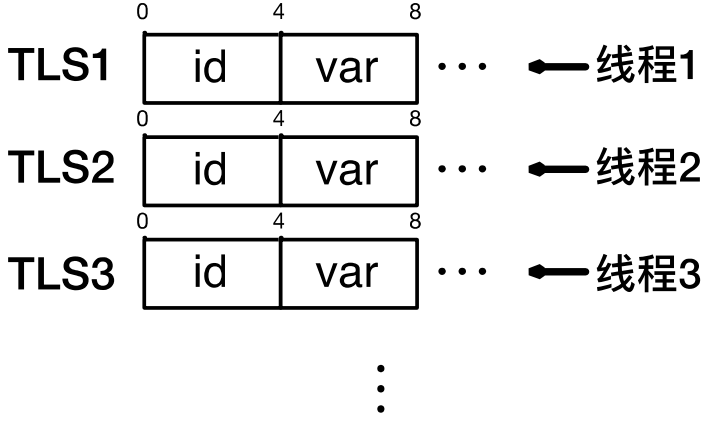
- **一对一模型的TCB可以分为两部分**
- **内核态：与PCB结构类似**
  - Linux中进程与线程使用的是同一种数据结构（task_struct）
  - 上下文切换中会使用
- **应用态：可以由线程库定义**
  - Linux：pthread结构体
  - Windows：TIB（Thread Information Block）
  - 可以认为是内核TCB的扩展

线程本地存储（TLS）

- **不同线程可能会执行相同的代码**
  - 线程不具有独立的地址空间，多线程共享代码段
- **问题：对于全局变量，不同线程可能需要不同的拷贝**
  - 举例：用于标明系统调用错误的errno
- **解决方案：线程本地存储（Thread Local Storage）**

线程本地存储（TLS）

- **线程库允许定义每个线程独有的数据**
  - `__thread int id;` 会为每个线程定义一个独有的id变量
- **每个线程的TLS结构相似**
  - 可通过TCB索引
- **TLS寻址模式：基地址 + 偏移量**
  - X86: 段页式 (fs寄存器)
  - RISC-V: 通用寄存器tp(x4)



线程的基本操作：以*pthreads*为例

- **Portable Operating System Interface, POSIX**
  - IEEE定义的标准，在不同OS维护应用的兼容性
  - 定义了应用程序编程接口（API）
    - 包括命令行shell、基础工具集
    - 在类Unix系统和其他系统间维护兼容性
- **POSIX Threads，Pthreads**
  - 线程的POSIX标准
  - 定义了创建和管理线程的API

线程的基本操作：以*pthreads*为例

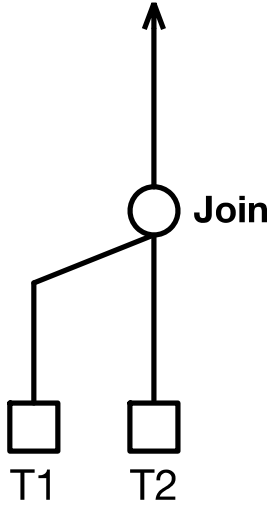
- **Pthreads**
  - 定义了一组C语言的数据结构类型、函数和约束
  - 头文件在操作系统定义的pthread.h
  - 库为的libpthread.so

- **分类**
  - 约100个Pthread相关API函数,都以pthread开头
  - 线程管理- creating, joining threads etc.
  - 互斥-Mutexes
  - 条件变量-Condition variables
  - 同步-线程间同步锁和barrier

线程的基本操作：以*pthreads*为例

- **创建：pthread_create**
  - 内核态：创建相应的内核态线程及内核栈
  - 应用态：创建TCB、应用栈和TLS

- **合并：pthread_join**
  - 等待另一线程执行完成，并获取其执行结果
  - 可以认为是fork的“逆向操作”



线程的基本操作：以*pthreads*为例

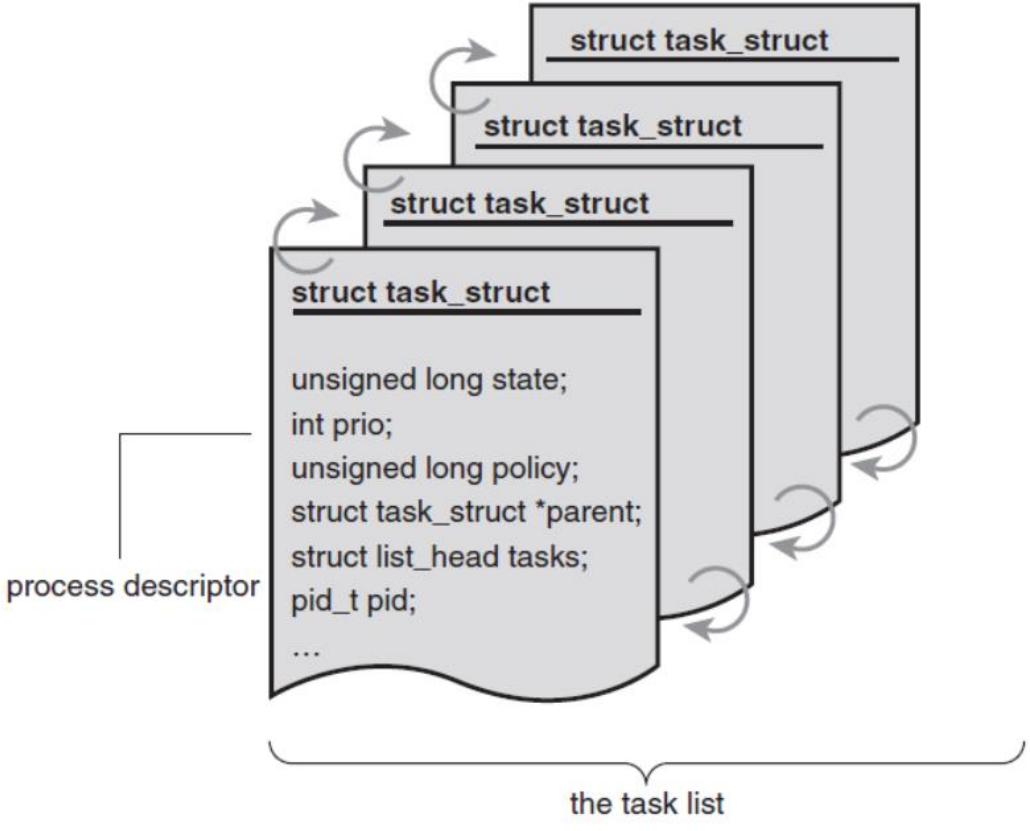
- **退出：pthread_exit**
  - 可设置返回值（会被pthread_join获取）

- **暂停：pthread_yield**
  - 立即暂停执行，出让CPU资源给其它线程
  - 好处：可以帮助调度器做出更优的决策

LINUX中线程管理

线程的概念 - 线程模型 - 相关数据结构 - 基本操作

Linux进程描述符

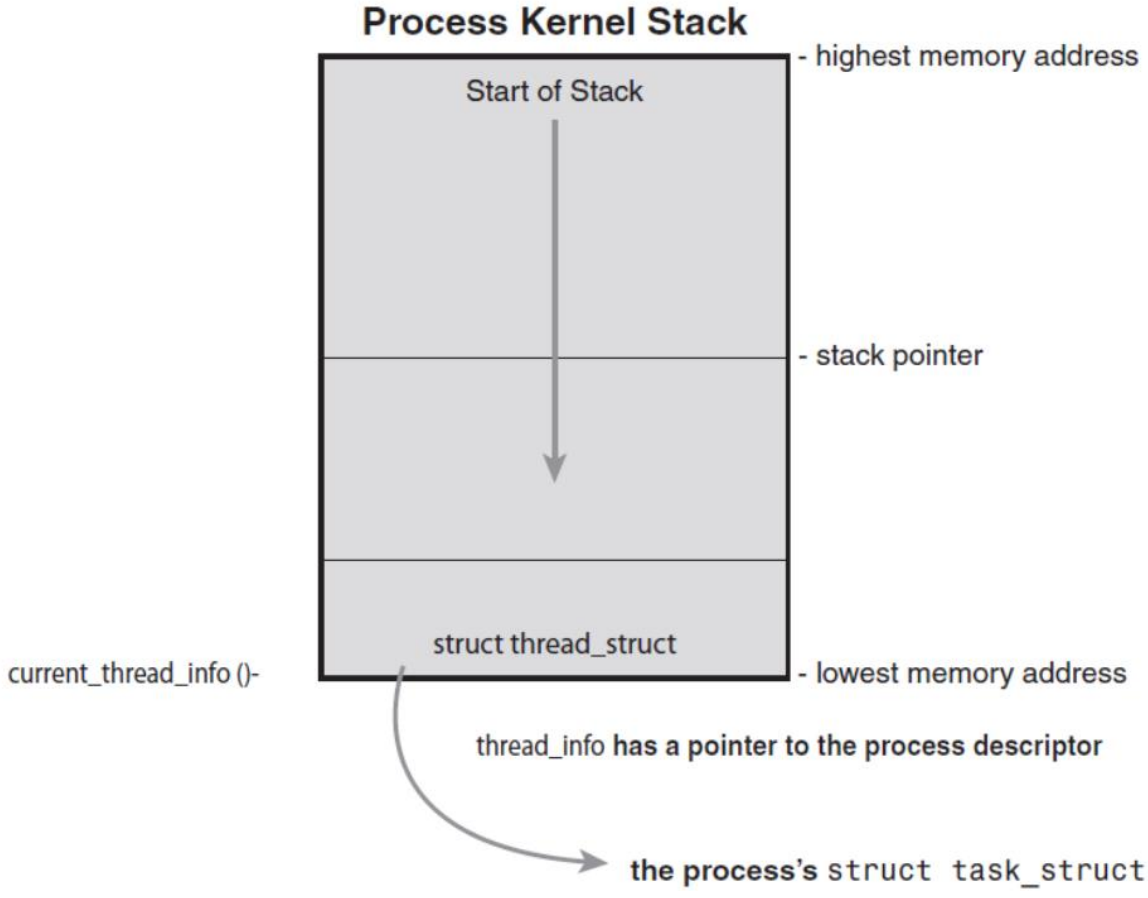

struct task_struct
struct task_struct
struct task_struct
struct task_struct
unsigned long state;
int prio;
unsigned long policy;
struct task_struct *parent;
struct list_head tasks;
pid_t pid;
…
process descriptor
the task list

Linux中的TCB

- Thread_info数据结构

```
struct thread_info {
        struct pcb_struct       pcb;            /* palcode state */

        struct task_struct      *task;          /* main task structure */
        unsigned int            flags;          /* low level flags */
        unsigned int            ieee_state;     /* see fpu.h */

        mm_segment_t            addr_limit;     /* thread address space */
        unsigned                cpu;            /* current CPU */
        int                     preempt_count; /* 0 => preemptable, <0 => BUG */
        unsigned int            status;         /* thread-synchronous flags */

        int bpt_nsaved;
        unsigned long bpt_addr[2];              /* breakpoint handling  */
        unsigned int bpt_insn[2];
};
```

进程描述和内核栈


Process Kernel Stack
Start of Stack
- highest memory address
- stack pointer
struct thread_struct
current_thread_info ()-
- lowest memory address
thread_info has a pointer to the process descriptor
the process's struct task_struct

Linux系统对线程的实现

- **Linux线程的独特实现**
  - 将所有的线程作为标准进程实现
  - 线程仅仅是一个与其它进程共享资源的进程
  - 每个线程有一个唯一的task_struct数据结构
- **创建线程的调用**
  - clone(CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGNHAND,0)
- **普通进程fork**
  - clone(SIGCHLD,0)
- **Vfork**
  - clone(CLONE_VFORK|CLONE_VM|SIGCHLD,0)

Clone调用的flags标记

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.
CLONE_CHILD_SETTID	Set the TID in the child.
CLONE_PARENT_SETTID	Set the TID in the parent.
CLONE_VM	Parent and child share address space.

Linux的内核线程

- **Linux kernel thread**
  - 在后台进行特定操作的线程，通常为守护线程 kernel daemon
  - 可正常被调度和抢占

- **与普通线程区别**
  - 内核线程没有自己的地址空间，只运行在内核态
  - 请注意与内核态线程模型的区别!

- **例子:**
  - 软中断线程ksoftirqd
  - flushd

Linux内核线程相关API

```
struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data,
                                   const char namefmt[],
                                   ...)

struct task_struct *kthread_run(int (*threadfn)(void *data),
                                void *data,
                                const char namefmt[],
                                ...)

int kthread_stop(struct task_struct *k)
```

上下文切换

上下文组成 – 上下文切换

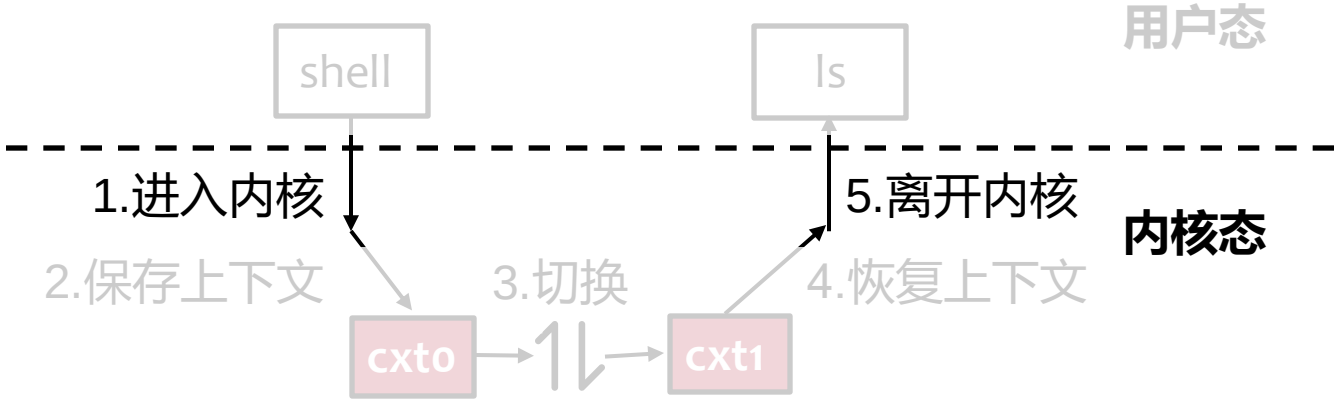
Linux Context Switch

- **切换代码在 kernel/sched.c, context_switch()**
- **内核代码在switch_to宏**
  - 切换代码都是架构相关的汇编指令
- **进程切换点在于栈指针的切换**
  - 当栈指针改变时，所有的局部变量随之改变

```
int global_val;   // 系统全局变量，进程间共享

void schedule()
{
  int tmp; //当前进程局部变量，存在于每个进程的堆栈上，进程间不可见
  ...
  context_switch();
  ...
```

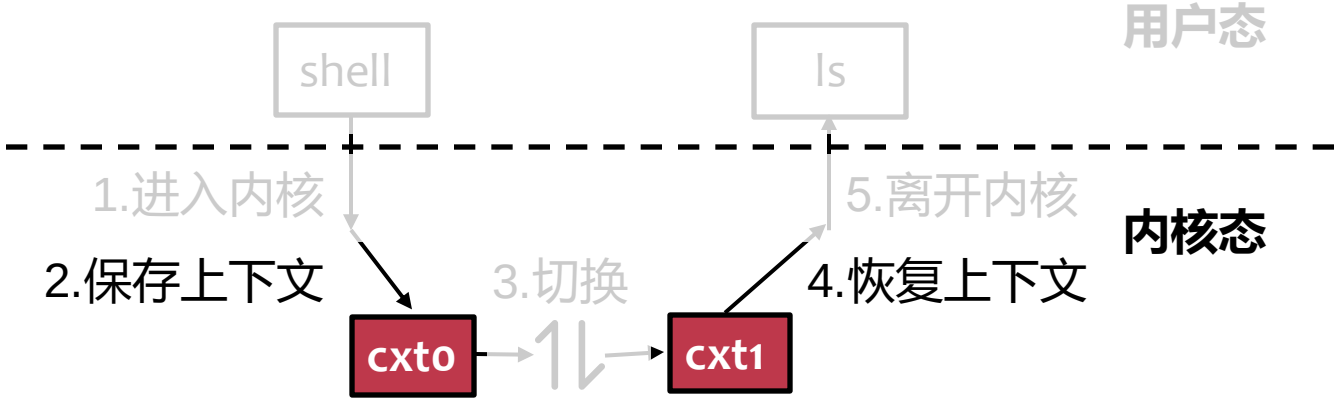
如何实现上下文切换?

- 进程怎样切换到内核中执行? 如何切换回用户态?



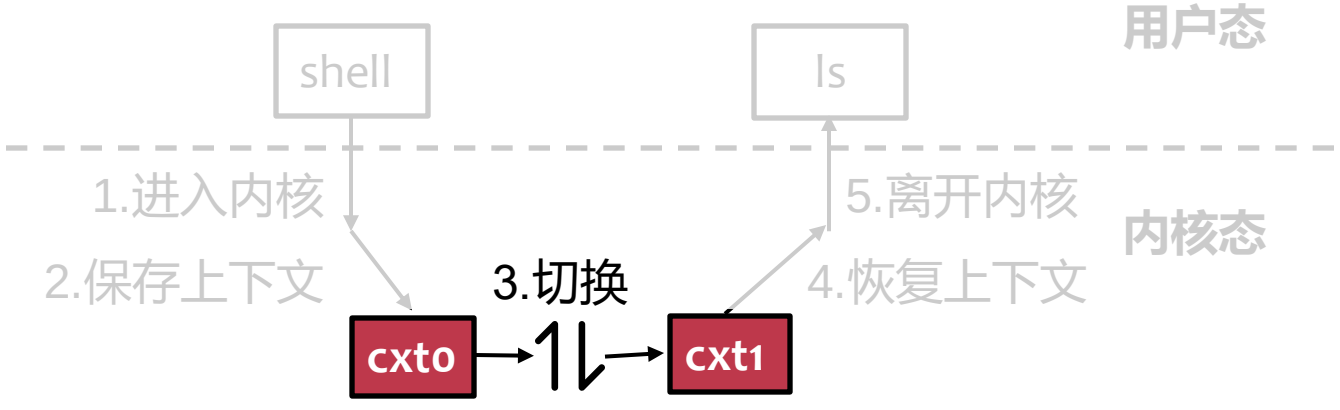
如何实现上下文切换?

- 进程怎样切换到内核中执行? 如何切换回用户态?
- 如何对上下文进行保存和恢复?



如何实现上下文切换?

- 进程怎样切换到内核中执行? 如何切换回用户态?
- 如何对上下文进行保存和恢复?
- 如何实现关键的切换步骤?



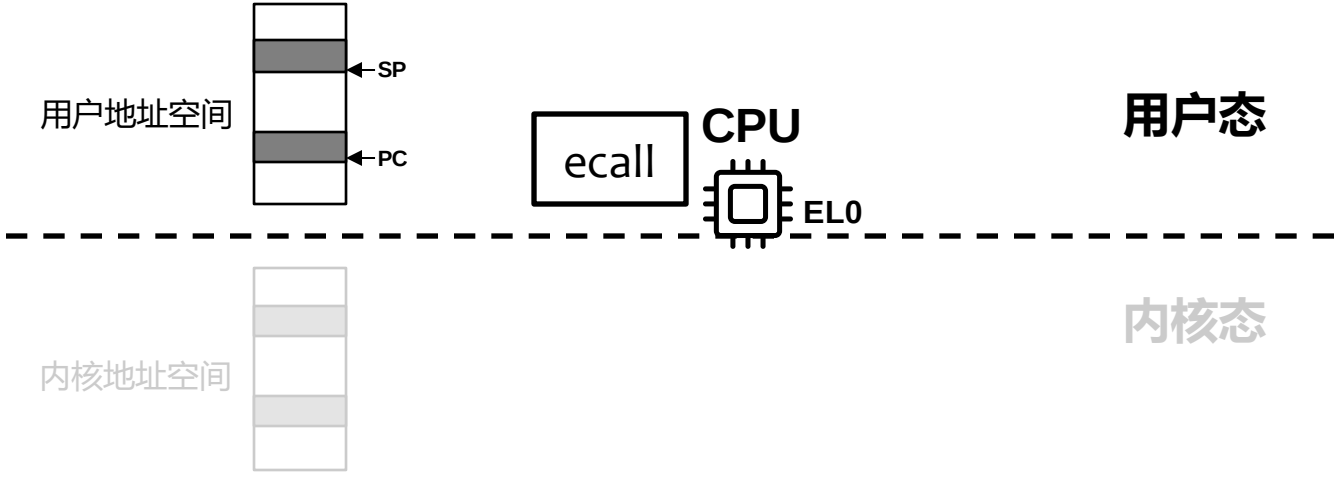
进程上下文的组成（RISC-V）

- **进程上下文需要包含哪些内容?**
  - 常规寄存器：X0-X31
  - 程序计数器（PC）
  - 系统寄存器，在不同的特权模式下有相对应的寄存器

- **思考：为什么进程上下文只需要保存寄存器信息，而不用保存内存?**

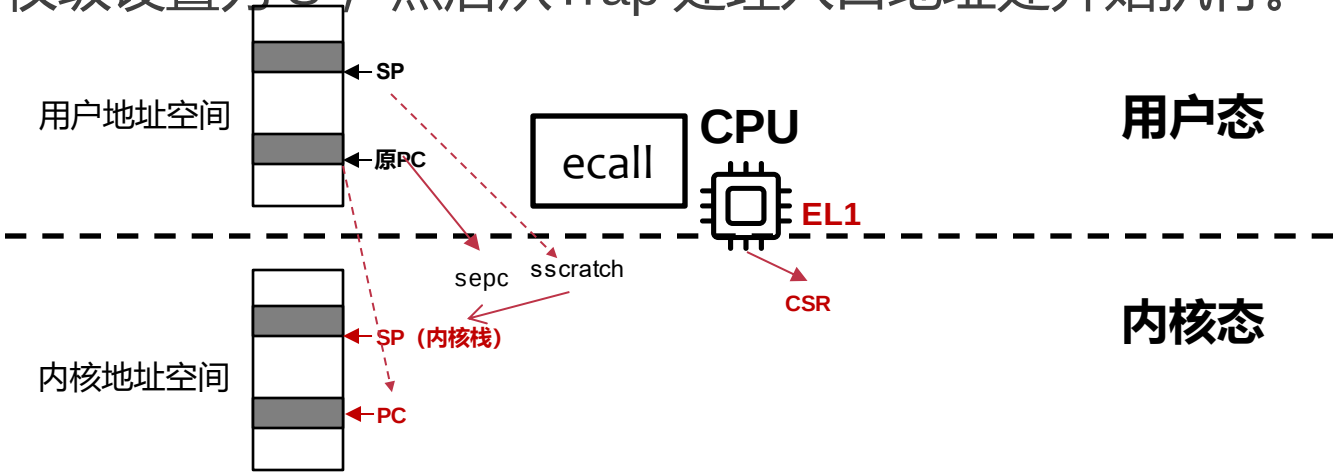
进程的内核态执行：切换到内核态

- RISC-V提供了硬件支持，使进程切换到内核态执行

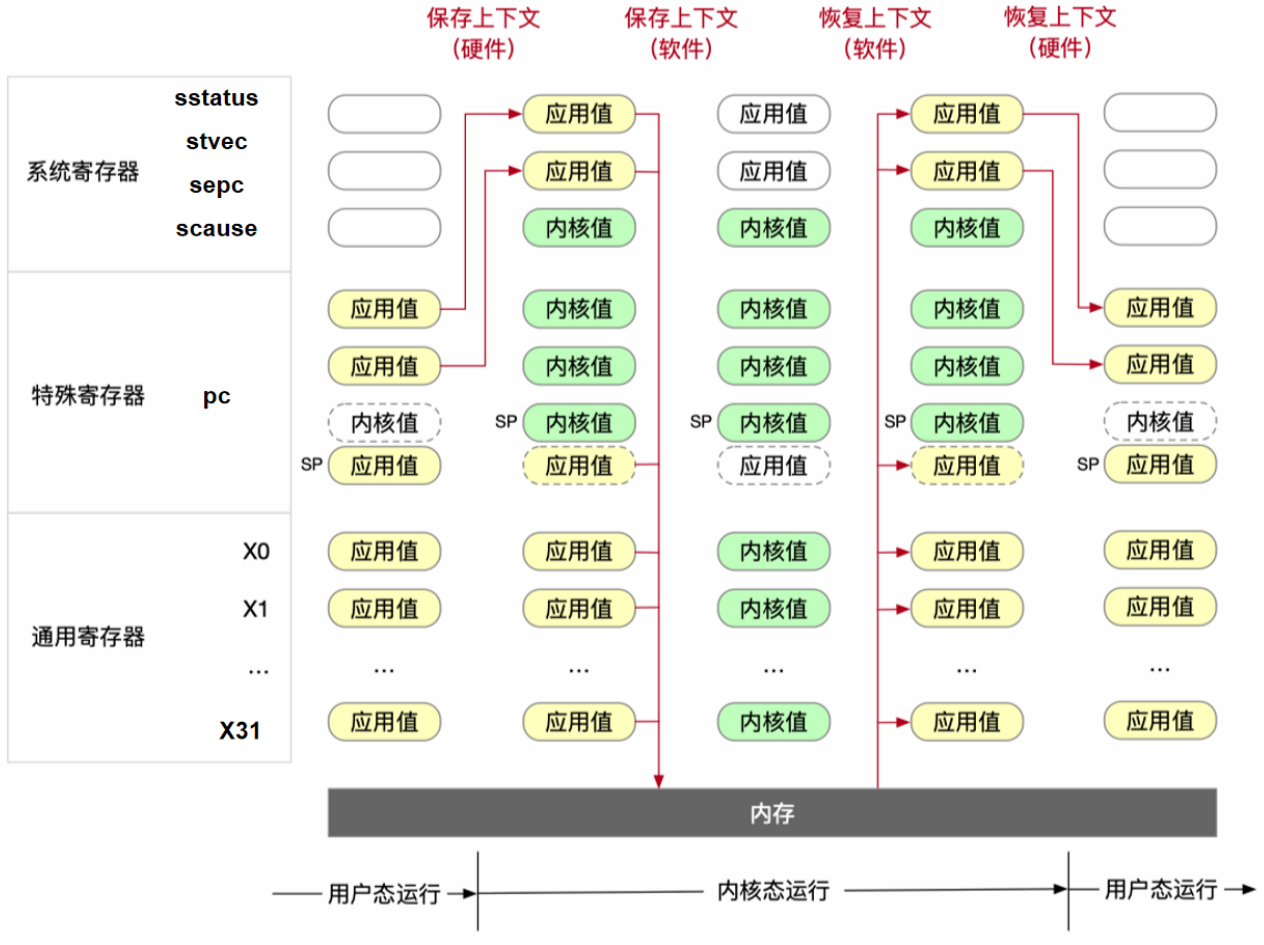


进程的内核态执行：切换到内核态

- **RISC-V提供了硬件支持，使进程切换到内核态执行**
  - 原特权级写入sstatus 的 SPP等字段
  - sepc 会被修改为 Trap 处理完成后会执行的下一条指令的地址。
  - scause/stval 分别会被修改成 Trap 的原因以及相关的附加信息。
  - CPU 会跳转到 stvec 所设置的 Trap 处理入口地址，并将当前特权级设置为 S，然后从Trap 处理入口地址处开始执行。



回顾：用户态/内核态切换时的处理器状态变化

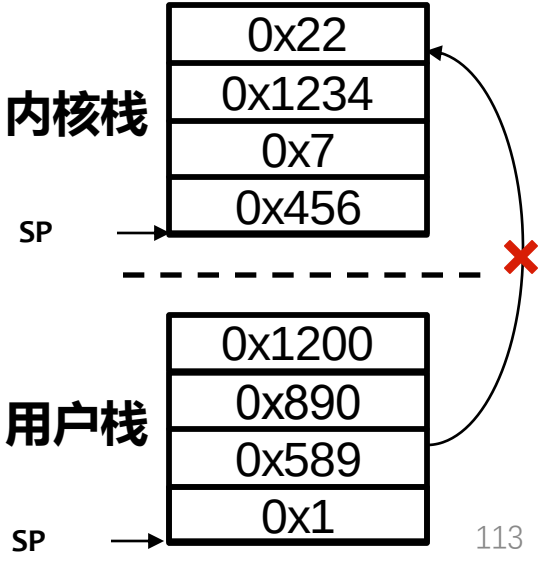


回顾：处理器在切换过程中的任务

1. **将发生异常事件的指令地址保存在sepc中**
2. **将异常事件的原因保存在scause/stval**
   - 例如，是执行svc指令导致的，还是访存缺页导致的
3. **将处理器的当前状态保存在sstatus寄存器的SPP 等字段**
4. **将引发缺页异常的内存地址保存在stvec中**
5. **栈寄存器不再使用SP（用户态栈寄存器），开始使用SP（内核态栈寄存器）**
   - 内核态栈寄存器，可以使用sscratch来保存当前进程task struct的地址，异常发生时取出
6. **修改sstatus寄存器中的特权级标志位，设置为内核态**
7. **找到异常处理函数的入口地址，并将该地址写入PC，开始运行操作系统**
   - 根据stvec寄存器中保存的异常向量表基地址，以及发生异常事件的类型确定

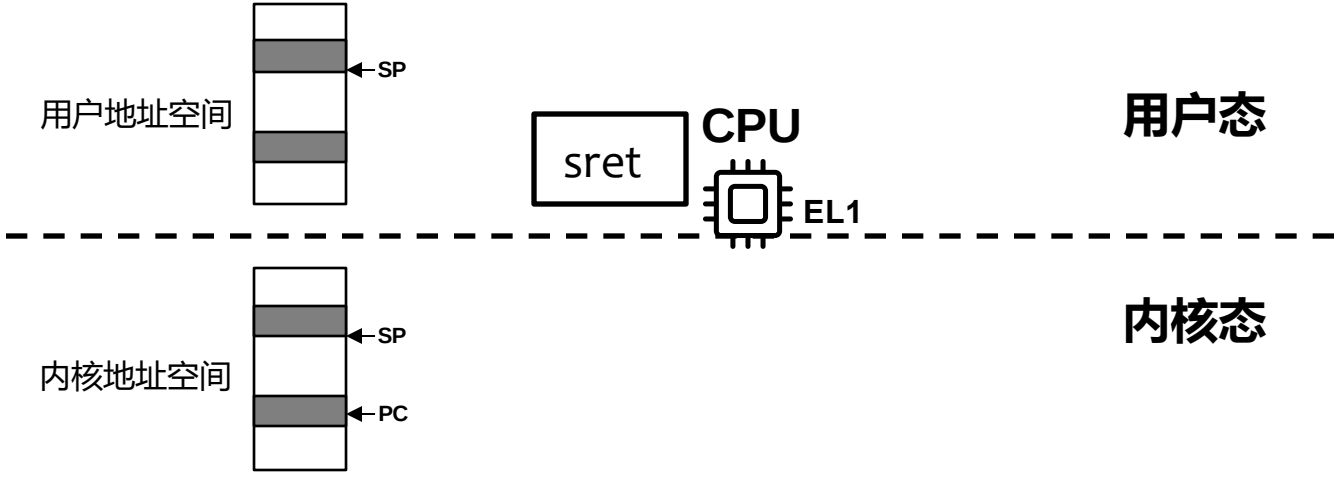
进程的内核态执行：内核栈

- **为什么需要“又一个栈”（内核栈）？**
  - 进程在内核中依然执行代码，有读写临时数据的需求
  - 进程在用户态和内核态的数据应该相互隔离，增强安全性
- **RISC-V实现：一个栈指针寄存器**
  - RISC-V只有通用寄存器SP，需要保存恢复
  - 从用户态进入时：使用sscratch存储内核栈地址作为中转



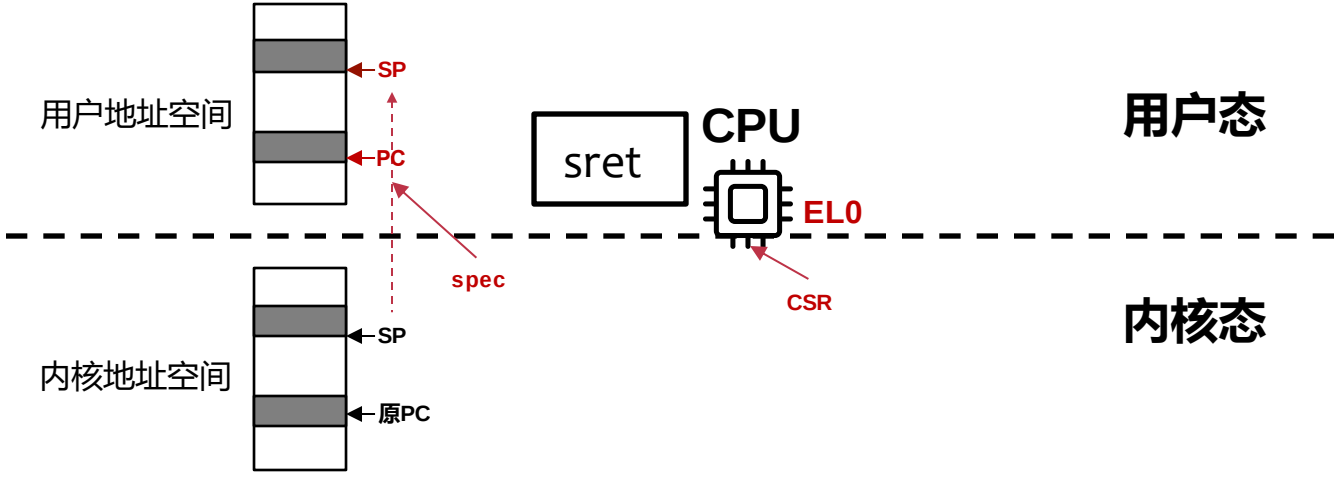
进程的内核态执行：返回用户态

- 进入内核态的“逆过程”，RISC-V同样提供了硬件支持



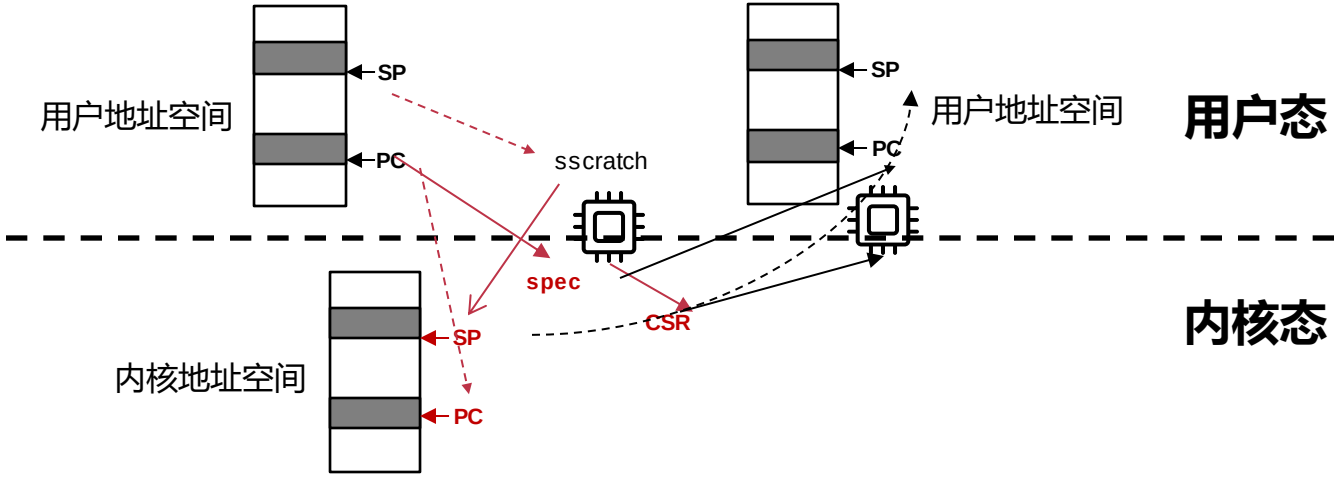
进程的内核态执行：返回用户态

- **进入内核态的“逆过程”，RISC-V同样提供了硬件支持**
  - CPU 会将当前的特权级按照 sstatus 的 SPP 字段设置为 U；
  - CPU 会跳转到 sepc 寄存器指向的那条指令，然后继续执行



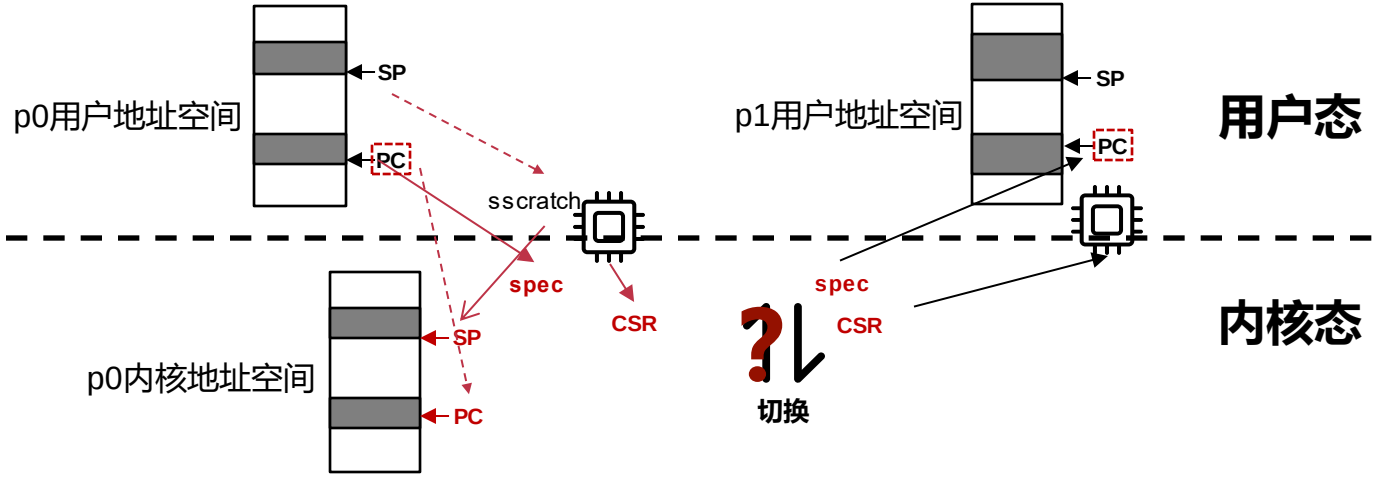
内核/用户态切换与上下文切换

- 通过支持内核/用户态切换，可以实现进程进入内核态并返回（比如system call）



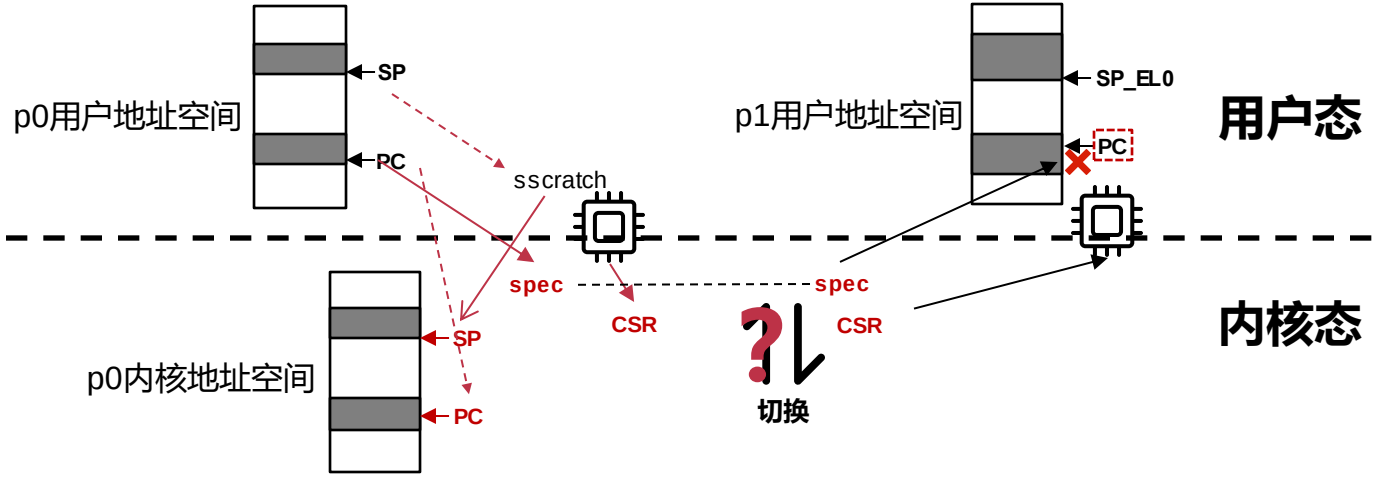
内核/用户态切换与上下文切换

- **但是该机制并不足以实现上下文切换**
  - 不同进程地址空间不同，使用的寄存器值也不同（如PC）



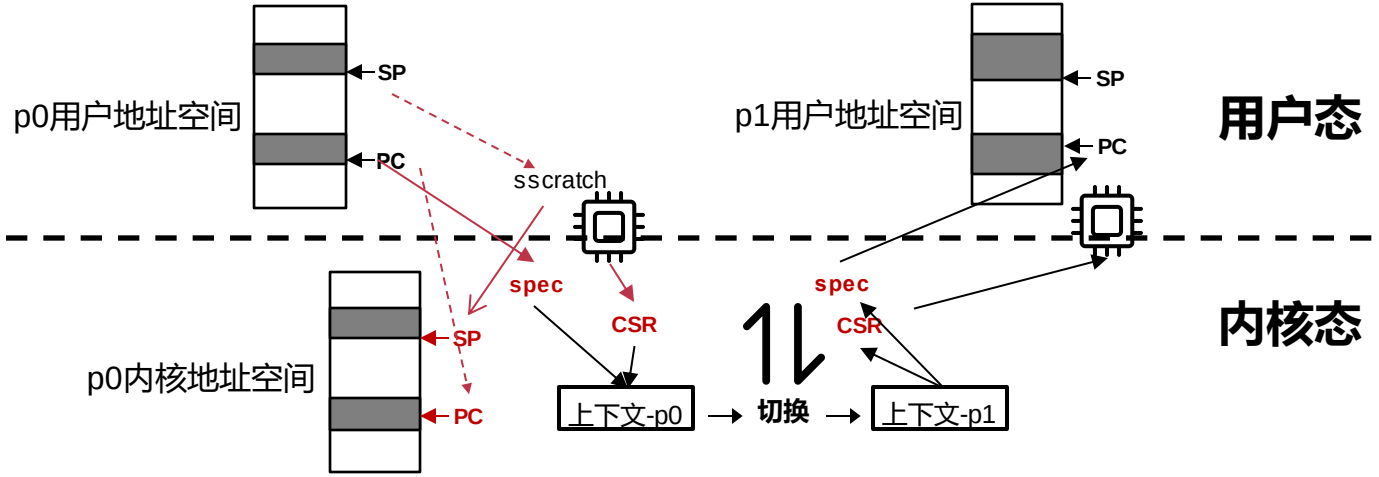
内核/用户态切换与上下文切换

- **但是该机制并不足以实现上下文切换**
  - 不同进程地址空间不同，使用的寄存器值也不同（如PC）
  - 但是：**寄存器只有一个！**直接恢复会导致错误



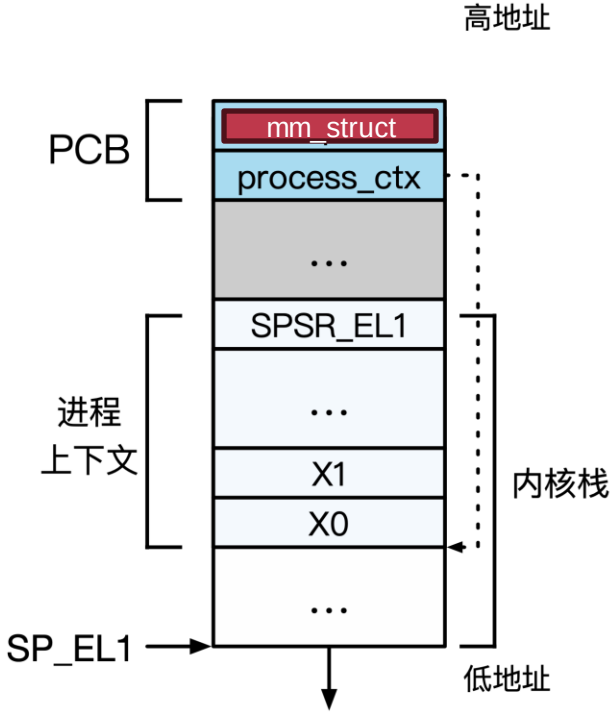
内核/用户态切换与上下文切换

- **但是该机制并不足以实现上下文切换**
  - 不同进程地址空间不同，使用的寄存器值也不同（如PC）
  - 但是：**寄存器只有一个！**直接恢复会导致错误
  - 解决方法：保存**上下文**（寄存器）到内存，用于之后恢复



上下文与其他内核数据结构

- 与进程相关的三种内核数据结构：PCB、上下文、内核栈
- PCB保存指向上下文的引用
- 上下文的位置**固定**在内核栈底部



上下文保存与恢复

发生异常时进入 handle_exception，第一步先保存上下文

```
SYM_CODE_START(handle_exception)
        /*
         * If coming from userspace, preserve the user thread pointer and load
         * the kernel thread pointer.  If we came from the kernel, the scratch
         * register will contain 0, and we should continue on the current TP.
         */
        csrrw tp, CSR_SCRATCH, tp
        bnez tp, _save_context

            ...

    _save_context:
            REG_S sp, TASK_TI_USER_SP(tp)
            REG_L sp, TASK_TI_KERNEL_SP(tp)
            addi sp, sp, -(PT_SIZE_ON_STACK)
            REG_S x1,  PT_RA(sp)
            REG_S x3,  PT_GP(sp)
            REG_S x5,  PT_T0(sp)
            save_from_x6_to_x31
```

```
/* save all GPs except x1 ~ x5 */
.macro save_from_x6_to_x31
REG_S x6,  PT_T1(sp)
REG_S x7,  PT_T2(sp)
REG_S x8,  PT_S0(sp)
REG_S x9,  PT_S1(sp)
REG_S x10, PT_A0(sp)
REG_S x11, PT_A1(sp)
REG_S x12, PT_A2(sp)
REG_S x13, PT_A3(sp)
REG_S x14, PT_A4(sp)
REG_S x15, PT_A5(sp)
REG_S x16, PT_A6(sp)
REG_S x17, PT_A7(sp)
REG_S x18, PT_S2(sp)
REG_S x19, PT_S3(sp)
REG_S x20, PT_S4(sp)
REG_S x21, PT_S5(sp)
REG_S x22, PT_S6(sp)
REG_S x23, PT_S7(sp)
REG_S x24, PT_S8(sp)
REG_S x25, PT_S9(sp)
REG_S x26, PT_S10(sp)
REG_S x27, PT_S11(sp)
REG_S x28, PT_T3(sp)
REG_S x29, PT_T4(sp)
REG_S x30, PT_T5(sp)
REG_S x31, PT_T6(sp)
.endm
```

arch/riscv/include/asm/asm.h

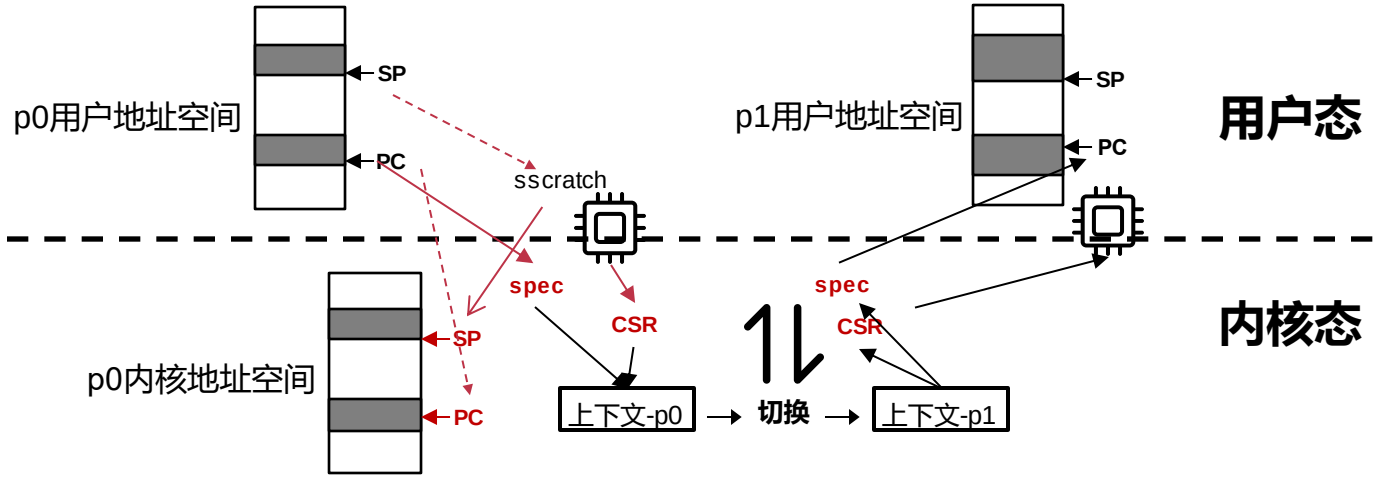
上下文保存与恢复

```
SYM_CODE_START_NOALIGN(ret_from_exception)
        REG_L s0, PT_STATUS(sp)

          ...
        csrw CSR_STATUS, a0
        csrw CSR_EPC, a2

        REG_L x1,  PT_RA(sp)
        REG_L x3,  PT_GP(sp)
        REG_L x4,  PT_TP(sp)
        REG_L x5,  PT_T0(sp)
        restore_from_x6_to_x31

        REG_L x2,  PT_SP(sp)


          ...
```

恢复上下文类似，只是反过来操作

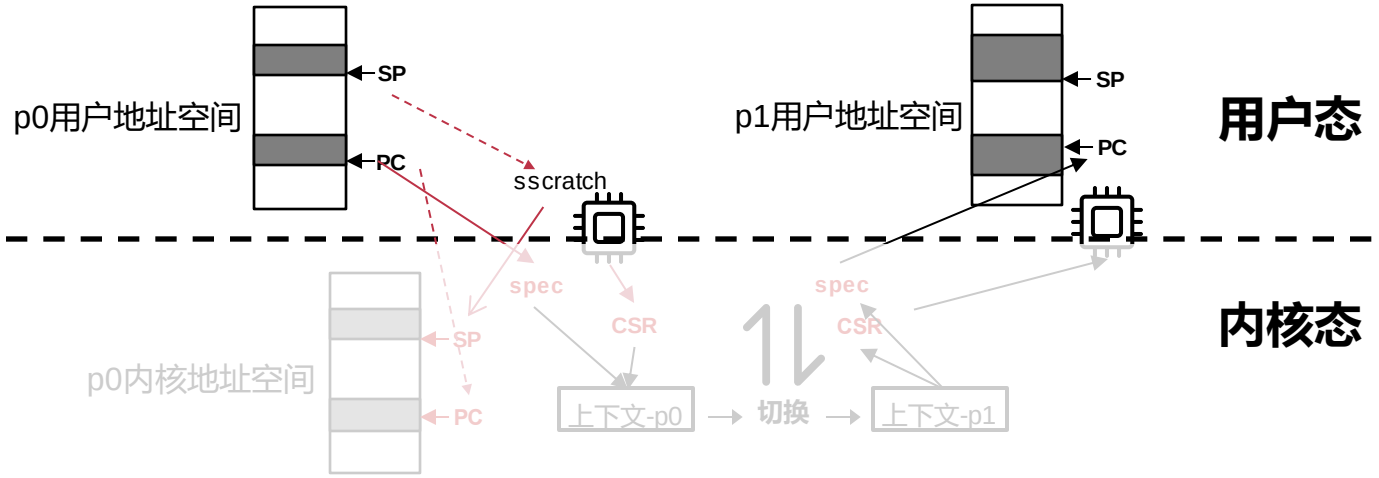
万事俱备，只欠切换！

- 上下文保存/恢复机制为进程间切换奠定了基础
- **思考：最关键的切换包含哪些步骤呢？**



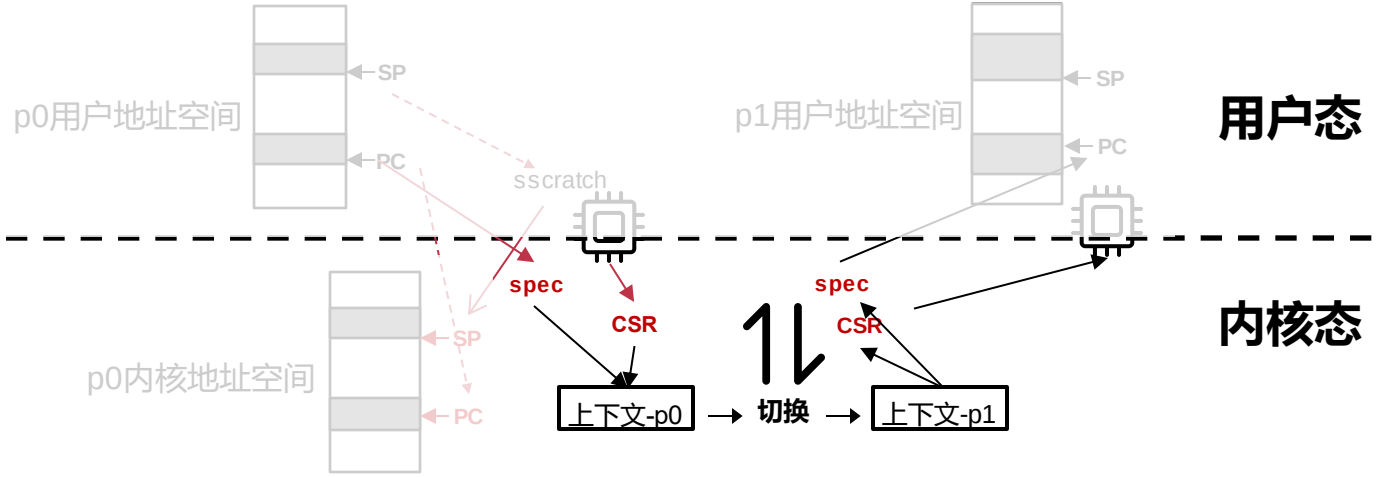
万事俱备，只欠切换！

- 上下文保存/恢复机制为进程间切换奠定了基础
- **思考：最关键的切换包含哪些步骤呢?**
  - 关键1:如何切换到p1的地址空间?



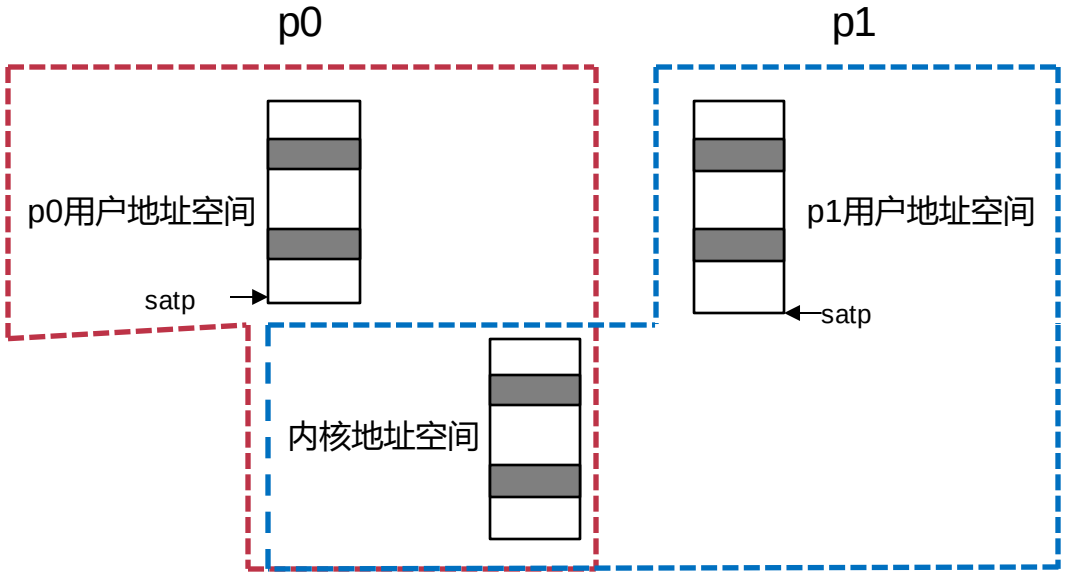
万事俱备，只欠切换！

- 上下文保存/恢复机制为进程间切换奠定了基础
- **思考：最关键的切换包含哪些步骤呢？**
  - 关键1:如何切换到p1的地址空间？
  - 关键2:如何切换到已经存储的p1上下文并进行恢复？



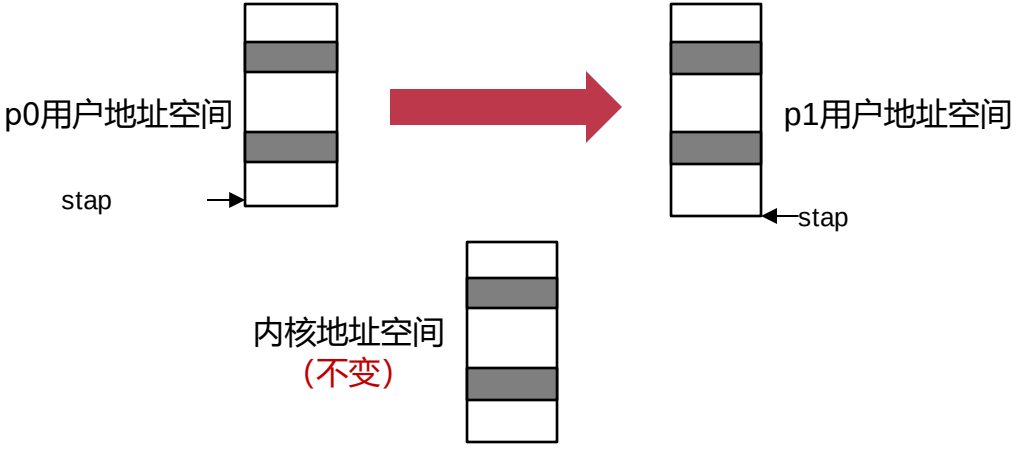
步骤1：地址空间的切换

- **回顾：RISC-V的地址空间管理**
  - 内核与用户态地址空间分开管理
  - 用户地址空间独有，内核地址空间共享



步骤1：地址空间的切换

- **回顾：RISC-V的地址空间管理**
  - 内核与用户态地址空间分开管理
  - 用户地址空间独有，内核地址空间共享
  - 因此，只需要实现用户地址空间切换即可



步骤1：地址空间的切换

- **RISC-V中的实现：switch_mm()**
  - 获取p1的PCB，并获取其mm_struct，最后设置satp

```
switch_mm(struct mm_struct *prev, struct mm_struct *next,
          struct task_struct *task)
```

```
static void set_mm_noasid(struct mm_struct *mm)
{
        csr_write(CSR_SATP, virt_to_pfn(mm->pgd) | satp_mode);
        local_flush_tlb_all();
}
```

转为物理地址

```
static inline void local_flush_tlb_all(void)
{
        __asm__ __volatile__ ("sfence.vma" : : : "memory");
}
```

刷新TLB（为什么？）

步骤2：如何切换到p1的上下文?

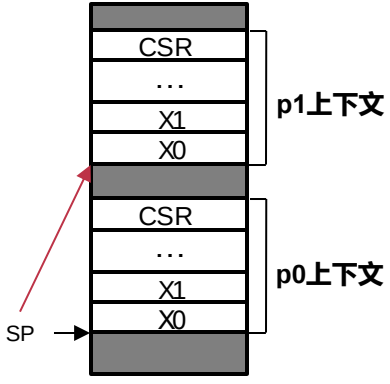
- 回顾：当p1保存上下文时，其内核栈应该是怎样的?

SP指向上下文起始地址

- p0/p1共享内核地址空间，因此直接切换内核栈指针即可

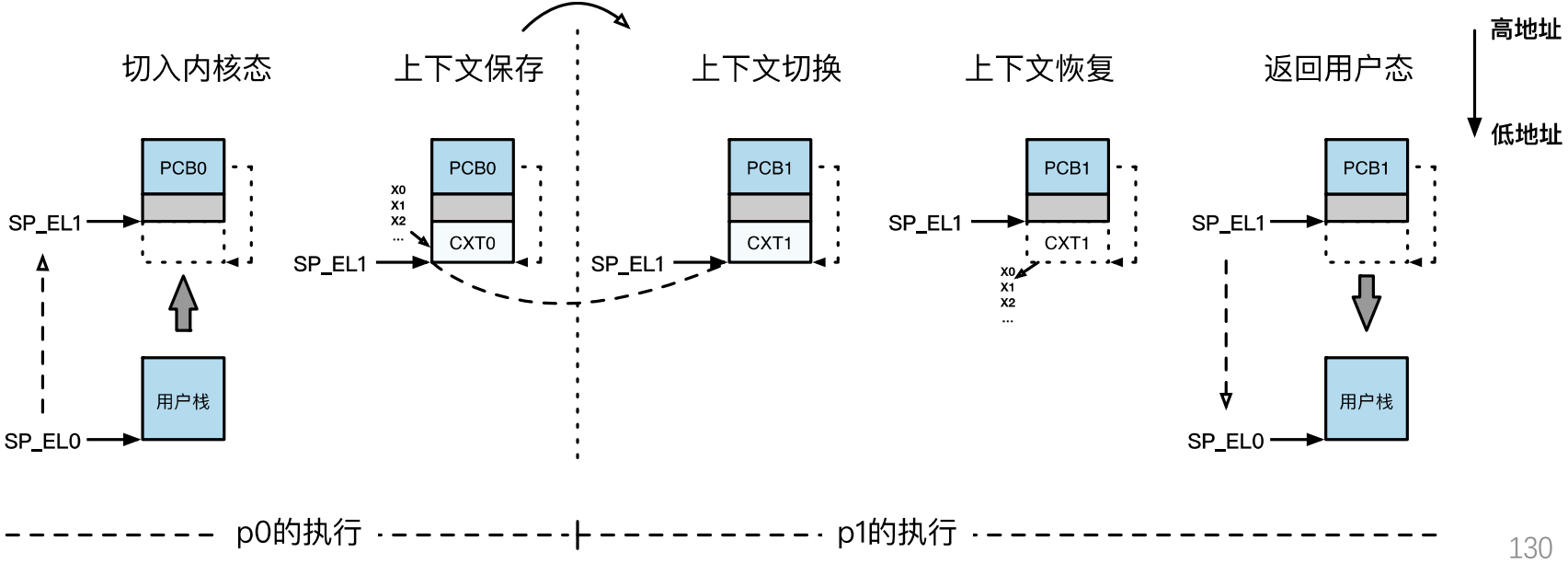
```
context_switch(struct rq *rq, struct task_struct *prev,
                    struct task_struct *next, struct rq_flags *rf){
     …
    /* Here we just switch the register
       state and the stack. */
     switch_to(prev, next, prev);
     barrier();
      …
}
```

```
    REG_L s11, TASK_THREAD_S11_RA(a4)
    /* The offset of thread_info in ta
    move tp, a1
    ret
SYM_FUNC_END(__switch_to)
```



总结：上下文切换栈变化全过程

- 共涉及两次权限等级切换、三次栈切换
- 内核栈的切换是线程切换执行的“分界点”



线程的上下文切换

- **保存上下文**
  - 寄存器信息（通用寄存器、浮点寄存器）
  - 协同处理器状态
  - 栈、Cache和TLB?
- **可能触发进程的上下文切换**
  - 什么时候触发?

纤程

纤程的概念 - 编程模型 - Windows和编程语言支持

一对一线程模型的局限

- **复杂应用：对调度存在更多需求**
  - 生产者消费者模型：生产者完成后，消费者最好马上被调度
  - 内核调度器的信息不足，无法完成及时调度

- **“短命”线程：执行时间亚毫秒级（如处理web请求）**
  - 内核线程初始化时间较长，造成执行开销
  - 线程上下文切换频繁，开销较大

纤程（用户态线程）

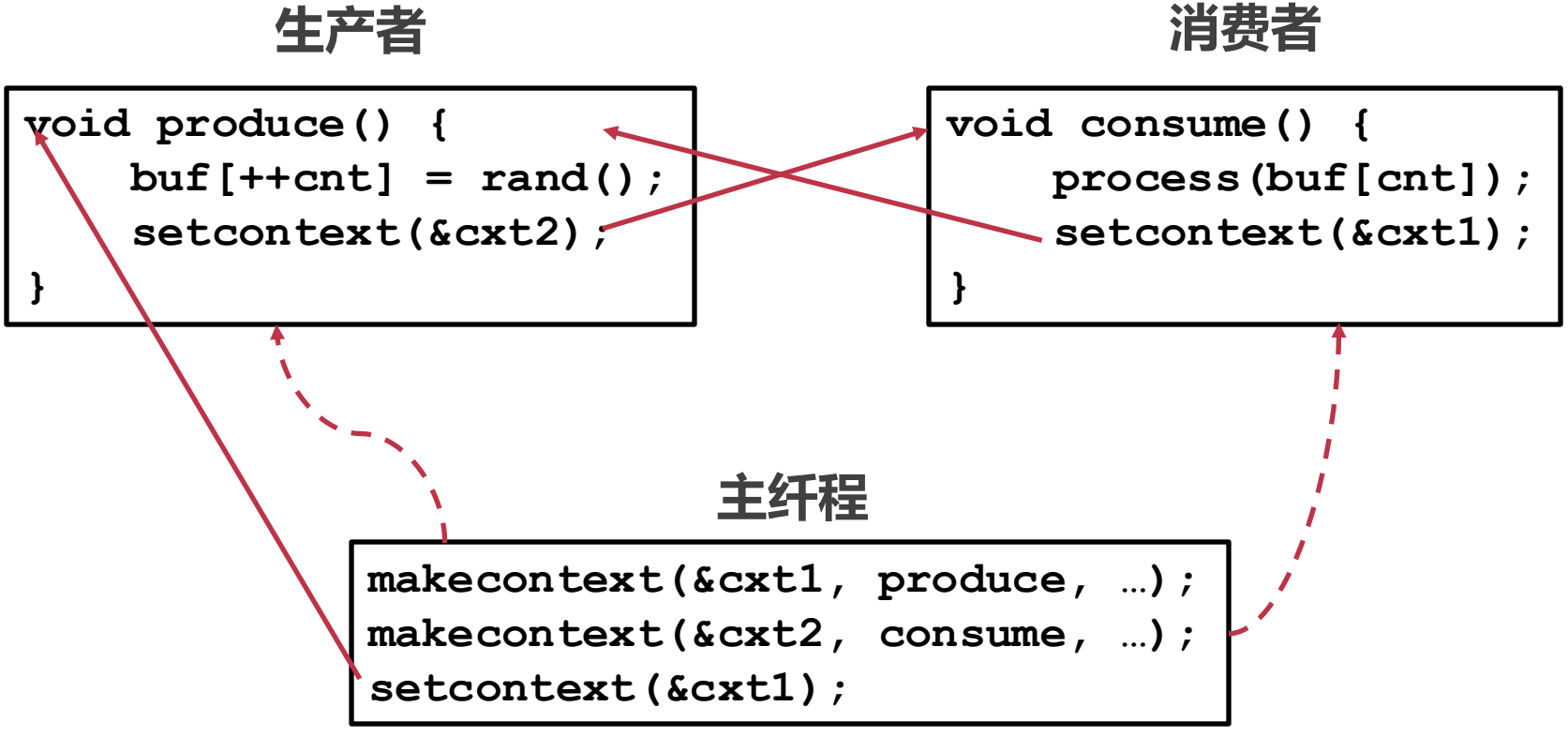
- **比线程更加轻量级的运行时抽象**
  - 不单独对应内核线程
  - 一个内核线程可以对应多个纤程（多对一）

- **纤程的优点**
  - 不需要创建内核线程，开销小
  - 上下文切换快（不需要进入内核）
  - 允许用户态自主调度，有助于做出更优的调度决策

Linux对于纤程的支持：ucontext

- **每个ucontext可以看作一个用户态线程**
  - makecontext: 创建新的ucontext
  - setcontext:纤程上下文切换
  - getcontext:保存当前的ucontext

纤程的例子：生产者 - 消费者



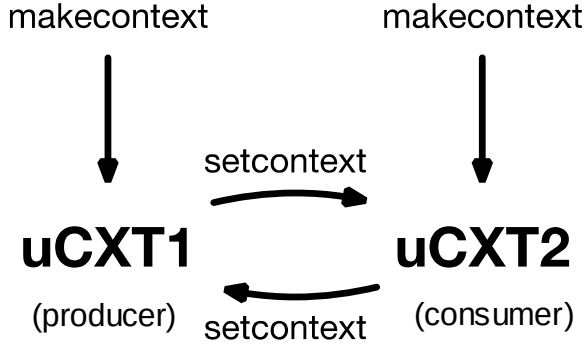
setcontext的代码片段

```
1  ENTRY (__setcontext)
2    ...
3    //恢复被调用者保存的通用寄存器
4    ldp x18, x19, [x0, register_offset + 18 * SZREG]
5    ldp x20, x21, [x0, register_offset + 20 * SZREG]
6    ldp x22, x23, [x0, register_offset + 22 * SZREG]
7    ldp x24, x25, [x0, register_offset + 24 * SZREG]
8    ...
9    //恢复用户栈
10   ldr x2, [x0, sp_offset]
11   mov sp, x2
12   //恢复浮点寄存器及参数
13   ...
14   //恢复 PC 并返回
15   ldr x16, [x0, pc_offset]
16   br x16
```

从例子看纤程的优势

- **纤程切换及时**
  - 当生产者完成任务后，可直接用户态切换到消费者
  - 对该线程来说是最优调度（内核调度器和难做到）

- **高效上下文切换**
  - 切换不进入内核态，开销小
  - 即时频繁切换也不会造成过大开销



Windows对于纤程的支持：Fiber库

- **与ucontext类似的编程模型**
  - createFiber: 创建新的纤程
  - SwitchToFiber: 纤程切换
- **支持纤程本地存储（FLS）**
  - Fiber Local Storage
  - 当一个内核线程对应单个纤程时，FLS与TLS结构相同
  - 当一个内核线程对应多个纤程时，TLS可分裂为多个FLS

程序语言中对纤程的支持：协程

- **许多高级程序语言都对协程提供了支持**
  - go、python、lua……
  - C++自20开始也支持了协程
- **协程也拥有状态（新生 / 暂停 / 终止 / 执行）**
  - 核心操作：yield（使协程暂停执行）、resume（继续执行）