空天信息安全与可信计算
教育部重点实验室

Linux分析与安全设计

第三章 Linux进程管理和沙箱机制

第三章 Linux进程管理和沙箱机制

- 基本概念
- 进程状态和状态转换
- 进程创建和终止
- 进程调度
- 进程同步和锁机制
- Linux沙箱机制

基本概念

什么是进程

进程定义

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。

进程是LINUX内核最基本的抽象之一

什么是进程

进程特征

- 动态性：进程的实质是进程实体的一次执行过程，有一定的生命期
- 并发性：指多个进程实体同存于内存中，且能在一段时间内同时运行。
- 独立性：传统OS中，进程实体是一个能独立运行、独立分配资源和独立接受调度的基本单位。
- 异步性：指进程按各自独立的、 不可预知的速度向前推进， 或说进程实体按异步方式运行。
- 结构特性：由程序段、相关的数据段和PCB三部分便构成了进程实体（进程映像）。

进程描述

PCB（进程控制块）：包括描述信息、控制信息、资源管理信息、CPU现场保护结构等，是系统感知进程存在的唯一实体。

```
struct task_struct {
        volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
        unsigned long flags; /* per process flags */
        mm_segment_t addr_limit; /* thread address space:
                        0-0xBFFFFFFF for user-thead
                        0-0xFFFFFFFF for kernel-thread */
        struct exec_domain *exec_domain;
        long need_resched;
        long counter;
        long priority;
        /* SMP and runqueue state */
        struct task_struct *next_task, *prev_task;
        struct task_struct *next_run,  *prev_run;
        ...
        /* task state */
        /* limits */
        /* file system info */
        /* ipc stuff */
        /* tss for this task */
        /* open file information */
        /* memory management info */
        /* signal handlers */
         ...
};
```

进程属性相关信息

- state成员：用来记录进程的状态。
- pid成员：这个是进程唯一的进程标识符（Process Identifier。
- flag成员：用来描述进程属性的一些标志位。
- exit_code和exit_signal成员：用来存放进程退出值和终止信号，这样父进程可以知道子进程的退出原因。
- pdeath_signal成员：父进程消亡时发出的信号。
- comm成员：存放可执行程序的名称。
- real_cred和cred成员：用来存放进程的一些认证信息。

进程标识符PID

- 进程标识符PID （Process Identifier）：用来识别进程的唯一号码
- PID的类型是一个int类型，所以默认最大值是32768
- 为了循环使用PID编号，内核使用bitmap机制来管理当前已经分配的PID编号和空闲的PID编号
- getpid()系统调用返回当前进程的tgid值而不是线程的pid值
- 系统调用gettid会返回线程的PID

调度相关的信息

- prio成员：保存着进程的动态优先级，是调度类考虑的优先级。
- static_prio成员：静态优先级。内核不存储nice值，取而代之的是static_prio。
- normal_prio成员：基于static_prio和调度策略计算出来的优先级。
- rt_priority成员：实时进程的优先级。
- sched_class成员：调度类。
- se成员：普通进程调度实体。
- rt成员：实时进程调度实体。
- dl成员：deadline进程调度实体。
- policy成员：进程的类型，比如普通进程还是实时进程。
- cpus_allowed成员：进程可以在哪几个CPU上运行。

进程之间的关系

- real_parent成员：指向当前进程的父进程的task_struct数据结构。
- children成员：指向当前进程的子进程的链表。
- sibling成员：指向当前进程的兄弟进程的链表。
- group_leader成员：进程组的组长。

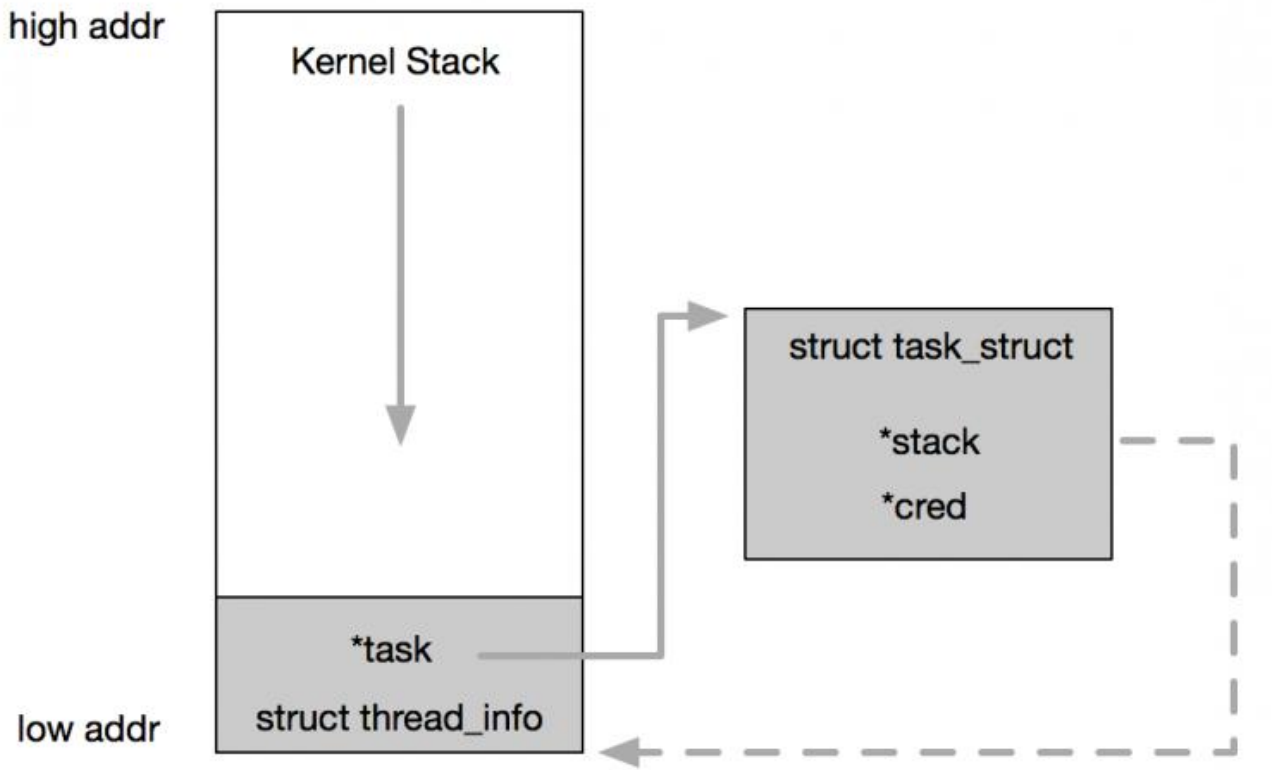
内存管理和文件管理相关信息

- mm成员：指向进程所管理的内存的一个总的抽象的数据结构mm_struct。
- fs成员：保存一个指向文件系统信息的指针。
- files成员：保存一个指向进程的文件描述符表的指针。

进程控制块总结

- 进程的运行状态
- 程序计数器
- CPU寄存器，保存上下文
- CPU调度信息
- 内存管理信息
- 统计信息
- 文件相关信息

内核栈和task_struct关系



- thread_info结构在内核栈尾端
- tack_struct中stack指向thread_info
- thread_info中task指向task_struct

thread_info

```
struct thread_info {
struct task_struct *task; /* main task structure */
struct exec_domain *exec_domain; /* execution domain */
  __u32 flags; /* low level flags */
  __u32 status; /* thread synchronous flags */
   __u32 cpu; /* current CPU */
  int saved_preempt_count;
   mm_segment_t addr_limit;
    struct restart_block restart_block;
  void __user *sysenter_return;
   #ifdef CONFIG_X86_32
  unsigned long previous_esp; /* ESP of the previous
stack in case of nested (IRQ) stacks */
  __u8 supervisor_stack[0];
  #endif
    unsigned int  sig_on_uaccess_error:1;
    unsigned int uaccess_err:1; /* uaccess failed */ };
```

获取当前进程task_struct

- **current**宏通过栈指针计算thread_info偏移，最后再从thread_info的task域中提取task_struct地址。

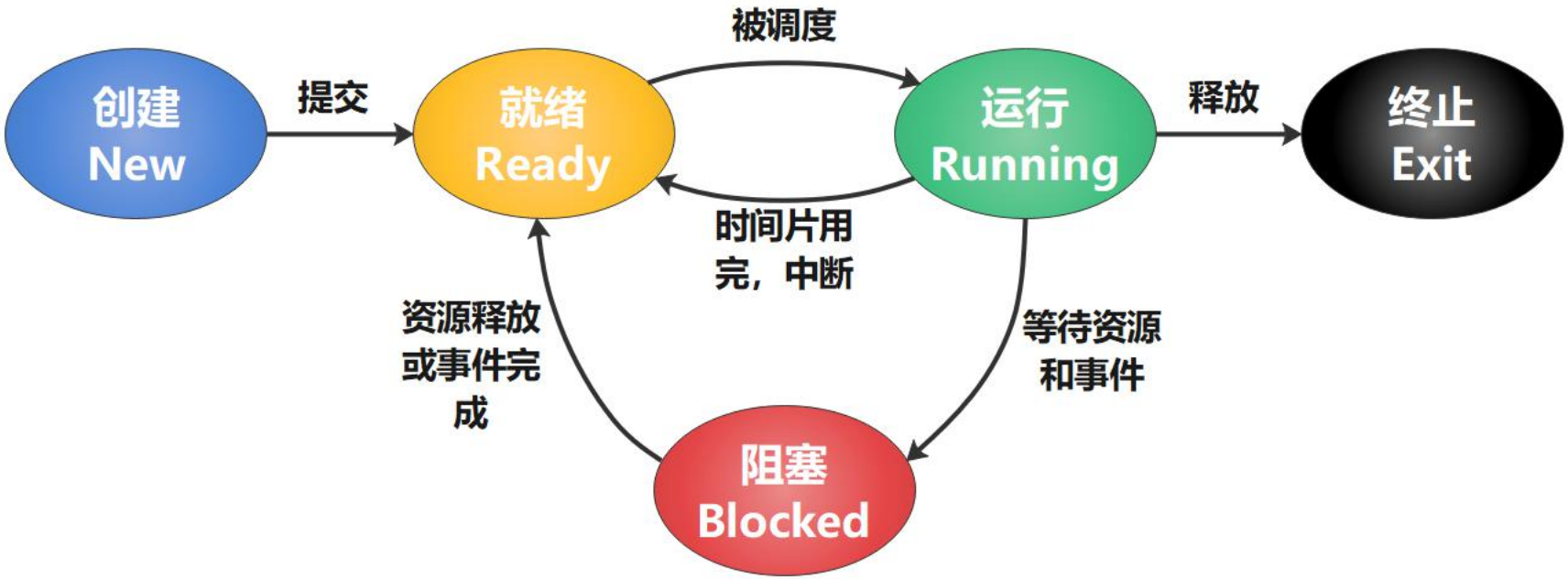
```
#define get_current() (current_thread_info()->task)
#define current get_current()

/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp") __used;

/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
        return (struct thread_info *)
                (current_stack_pointer & ~(THREAD_SIZE - 1));
}
```

进程状态和状态转换

进程状态

- 进程间经常需要相关沟通和交流，比如文本进程需要等待键盘的输入
- 进程状态包括：
  - ✓ 创建态：创建了新进程。
  - ✓ 就绪态：进程获得了可以运作的所有资源和准备条件。
  - ✓ 执行态：进程正在CPU中执行。
  - ✓ 阻塞态：进程因为等待某项资源而被暂时踢出了CPU。
  - ✓ 终止态：进程消亡

经典操作系统五状态模型



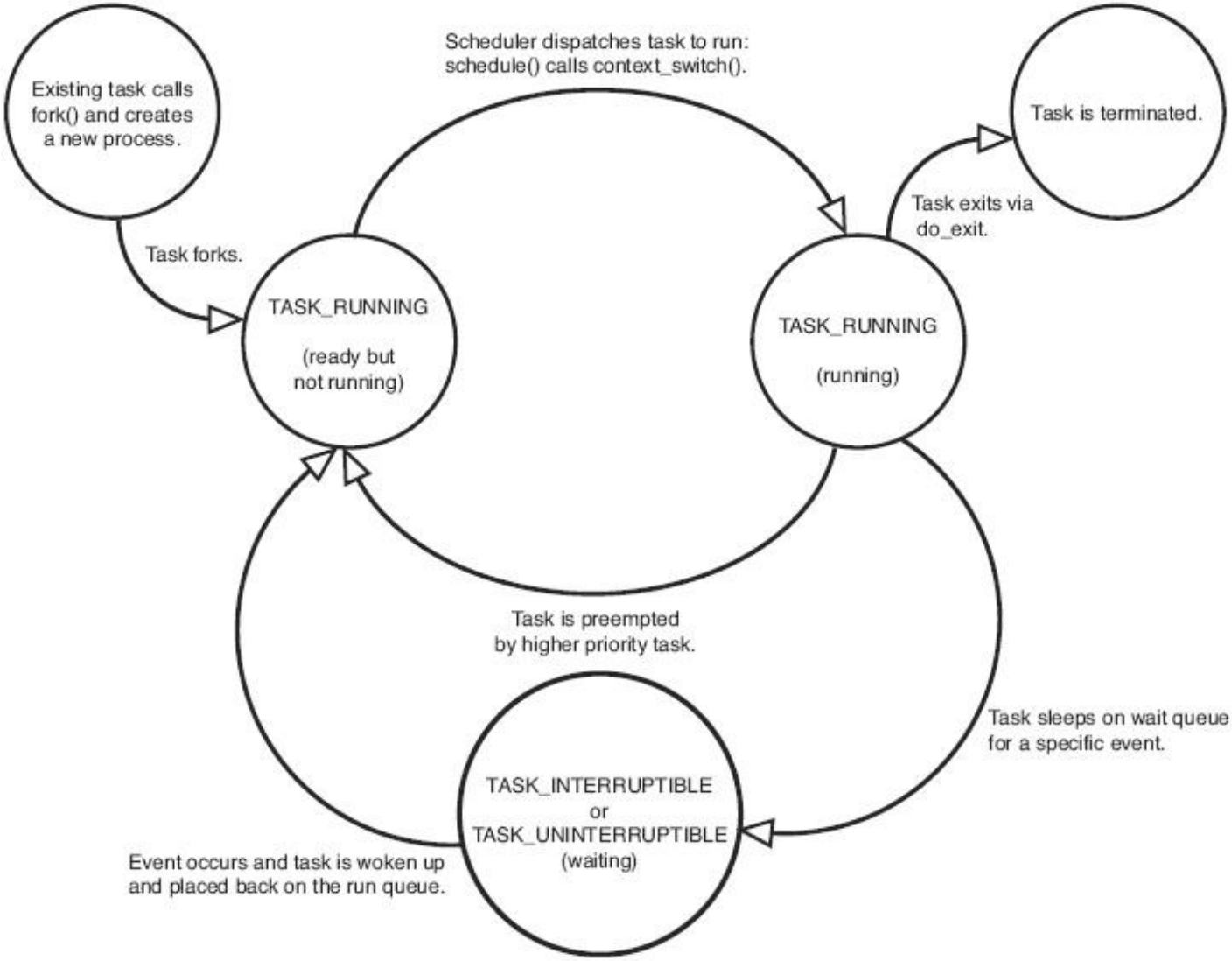
进程状态转换及原因

状态转换	原因
就绪—运行	进程被调度程序选中占用CPU。
运行—阻塞	进程让出CPU，等待系统分配资源或某些事件的发生，如：暂时不能访问某一资源，操作系统尚未完成服务，系统正在初始化I/O设备，等待用户的输入信息等。
阻塞—就绪	处于等待队列中的进程，当其等待的事件已经发生，或等待的资源可用时，此进程将进入就绪队列竞争CPU。
运行—就绪	进程分配的时间片已用完，或者在中断机制下，有更高优先级的进程进入系统，这时进程进入就绪队列等待下一次被选中而占用CPU。

Linux内核进程状态

进程状态	说明
TASK_RUNNING	可运行状态。未必正在使用CPU，也许是在等待调度
TASK_INTERRUPTIBLE	可中断的睡眠状态。正在等待某个条件满足
TASK_UNINTERRUPTIBLE	不可中断的睡眠状态。不会被信号中断
__TASK_STOPPED	暂停状态。收到某种信号，运行被停止
__TASK_TRACED	被跟踪状态。进程停止，被另一个进程跟踪
EXIT_ZOMBIE	僵尸状态。进程已经退出，但尚未被父进程或者init进程收尸
EXIT_DEAD	真正的死亡状态

Linux内核的进程状态转换图



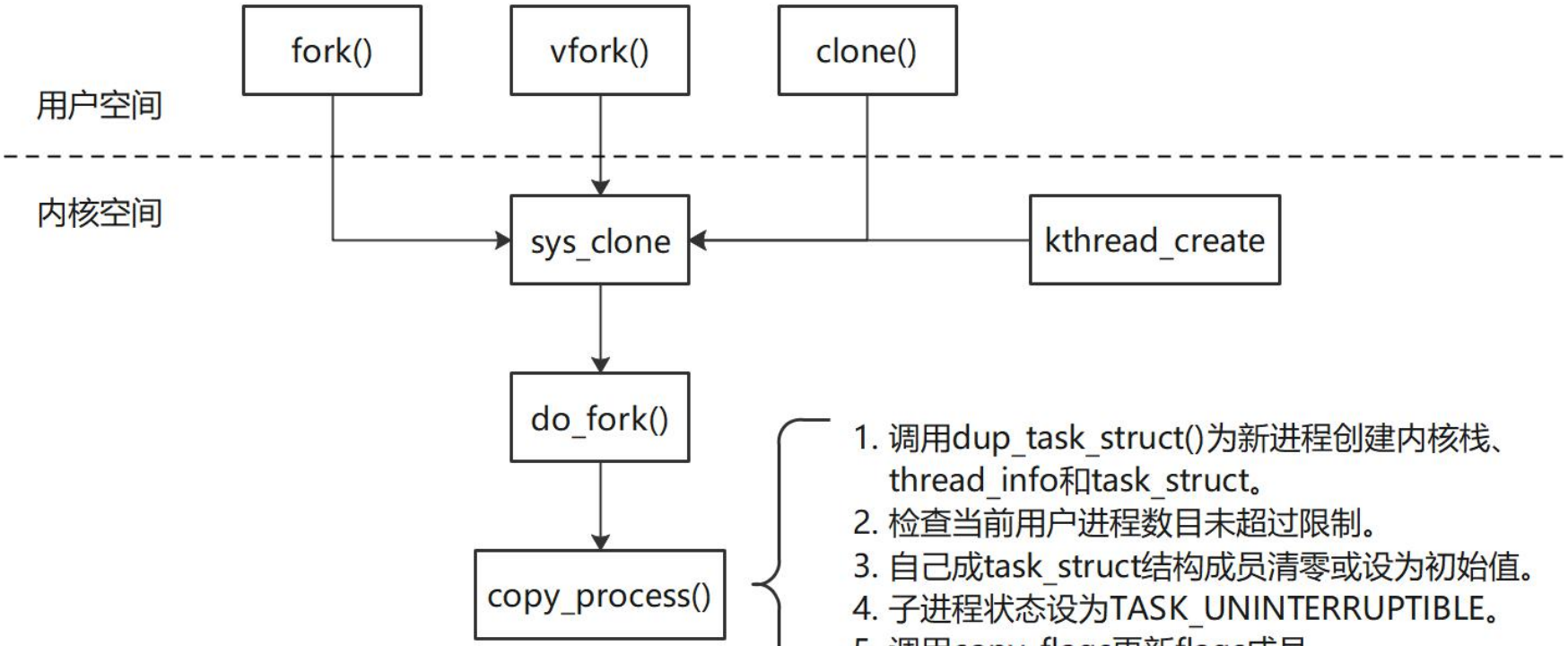
进程创建和终止

fork

```
算法fork
输入：无
输出：对父进程为子进程号
      对子进程为0
{
      检查可提供的核心资源；
      得到空闲进程表项和唯一的PID号；
      检查用户没有运行太多的进程；
      将子进程的状态标记为“被创建”；
      将父进程的进程表项中数据拷贝到子进程表项中；
      增加当前目录i结点的引用计数；
      在内存中拷贝父进程的环境（U区，正文、数据、栈）；
      If（运行进程是父进程）
      {
            将子进程的状态改变为“准备运行”状态；
            返回（子进程的ID）；
      } else     /*运行进程是子进程*/
      {
            初始化u区的计时字段；
            返回（0）；
      }
}
```

fork调用路径

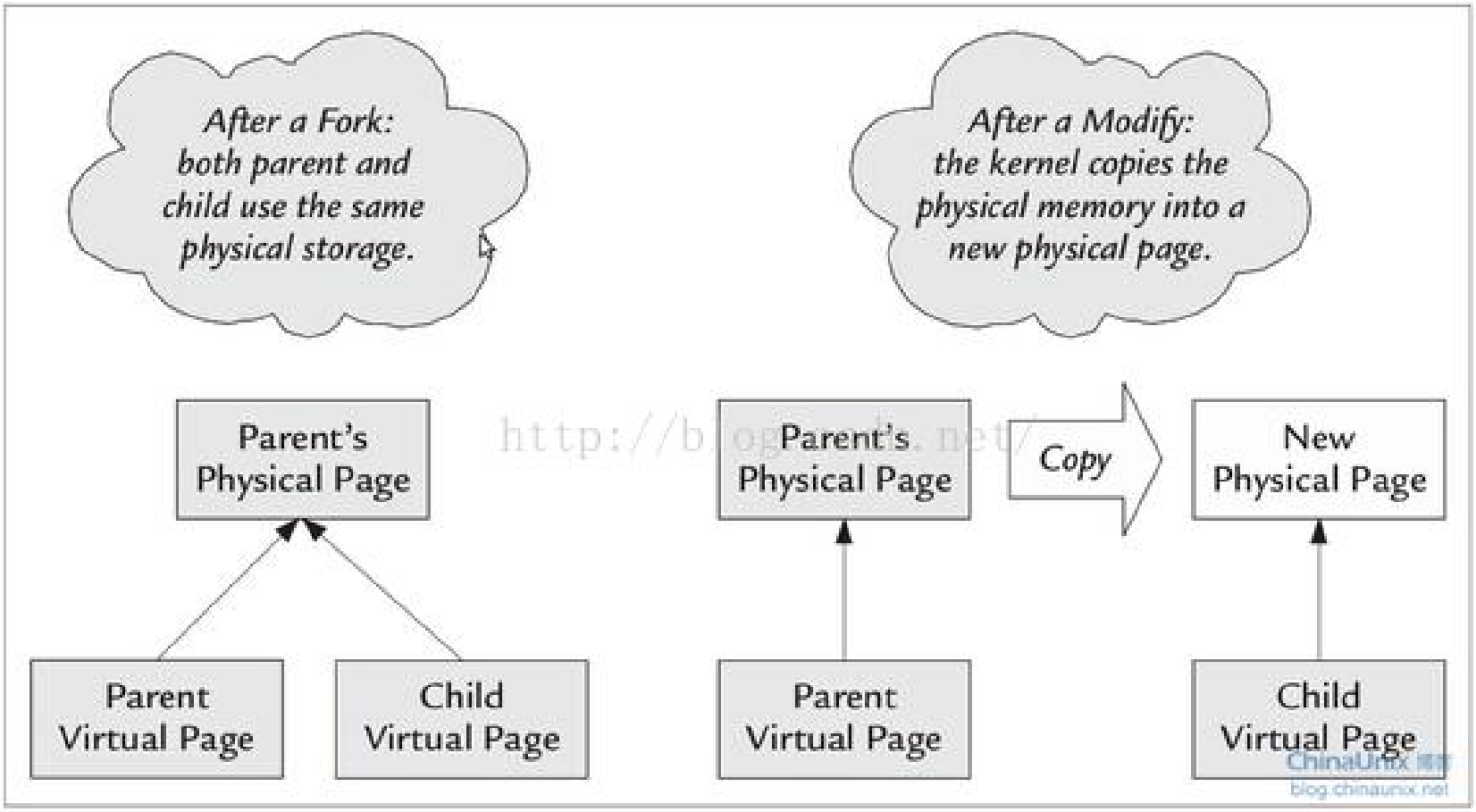
- 调用路径



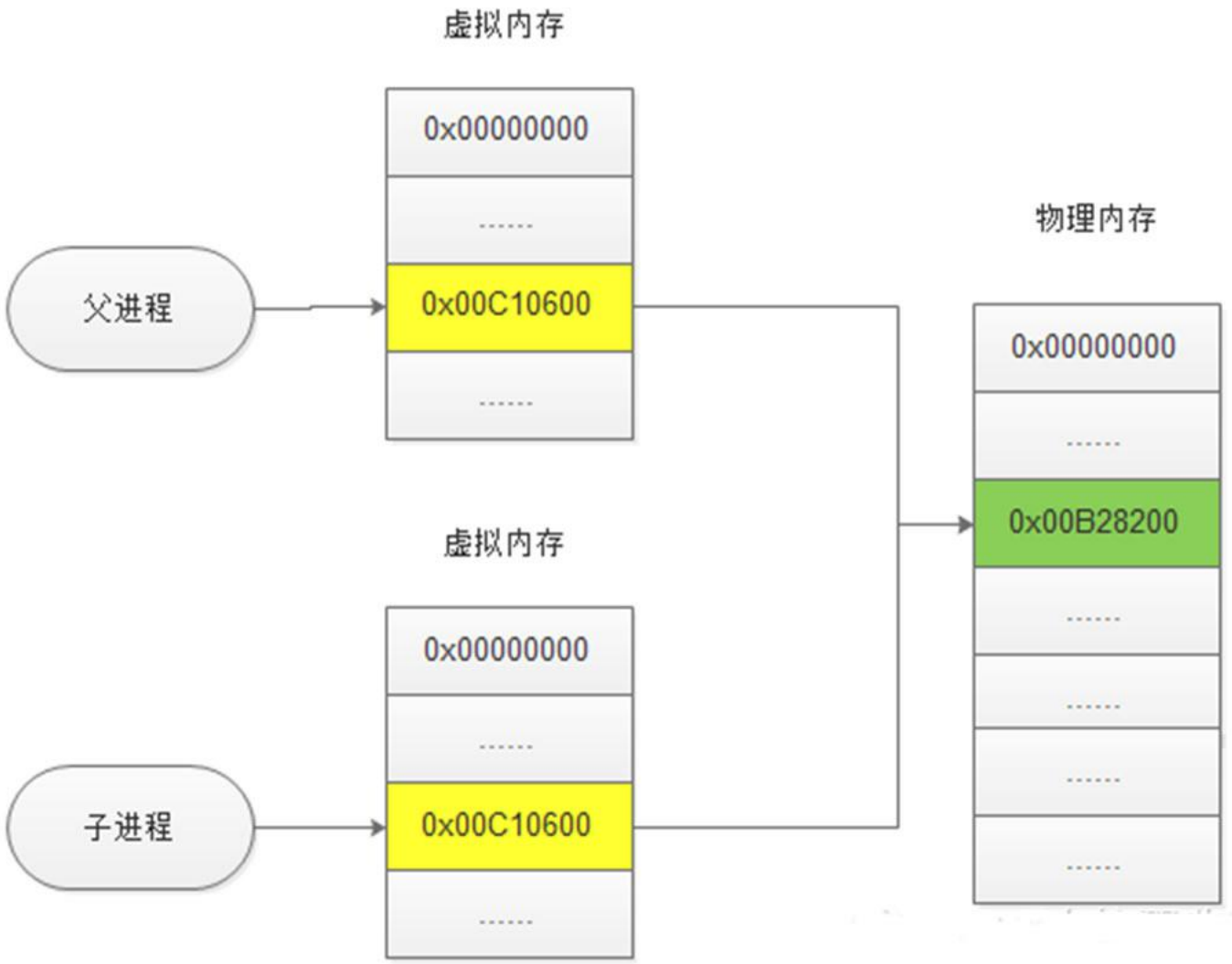
写时复制(COW)

- 写时复制技术（copy on write），简称CoW
- 写时复制技术就是父进程在创建子进程的时候，不需要拷贝进程地址空间的内容给子进程，只需要拷贝父进程的进程地址空间的页表给子进程即可，这样父子进程就共享了相同的进程地址空间。
- 写时复制技术使页的拷贝推迟到实际发生写入的时候。

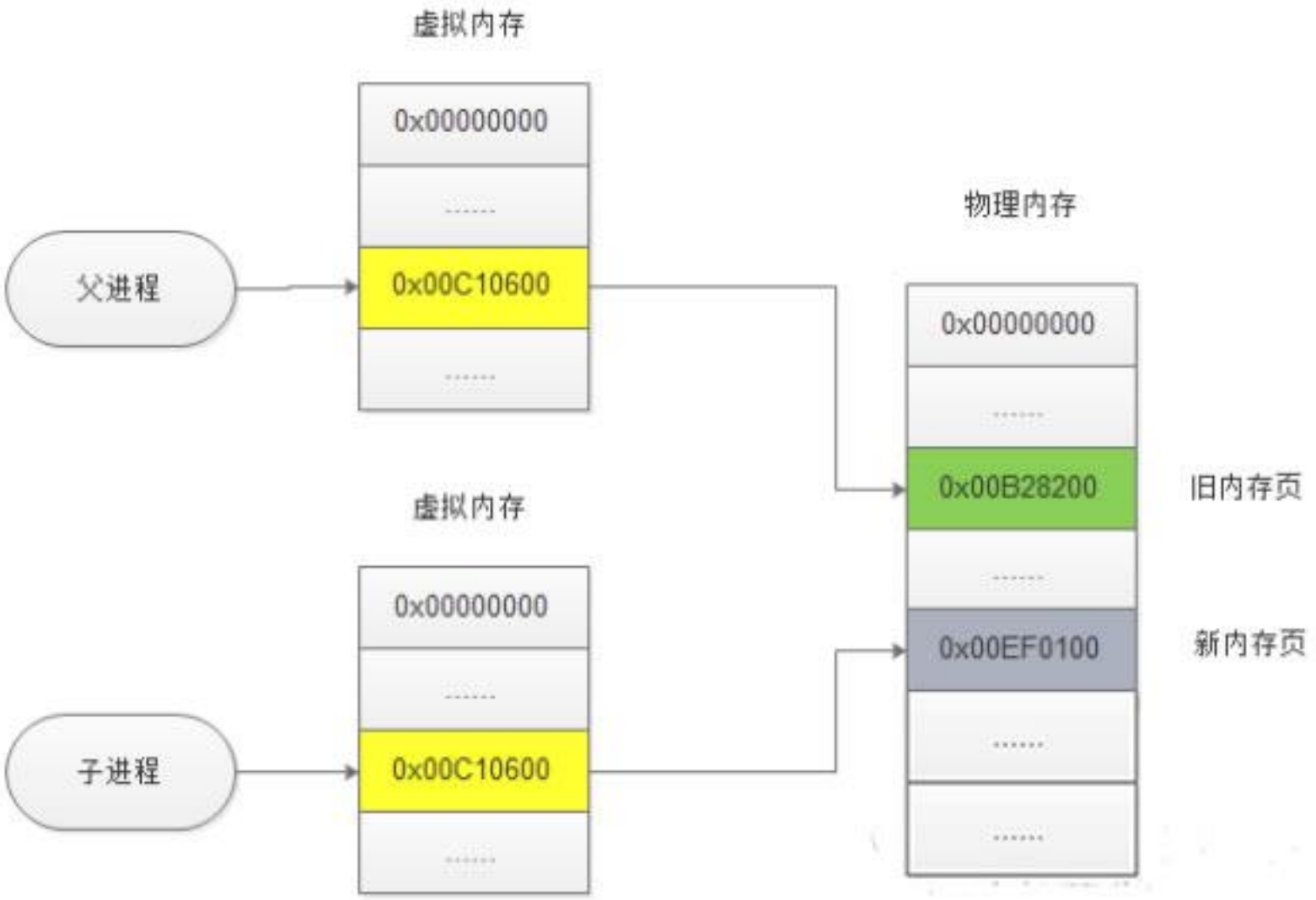
写时复制(COW)



写时复制(COW)



写时复制(COW)



vfork

- vfork：父进程会一直阻塞，直到子进程调用exit()或者exec()为止
- vfork()函数通过系统调用进入到Linux内核，然后通过do_fork()函数来实现。

```
/* Clone the calling process, but without copying the whole address space.
   The calling process is suspended until the new process exits or is
   replaced by a call to `execve'.  Return -1 for errors, 0 to the new process,
   and the process ID of the new process to the old process.  */
```

```
SYSCALL_DEFINE0(vfork)
{
        return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
                        0, NULL, NULL, 0);
}
```

clone

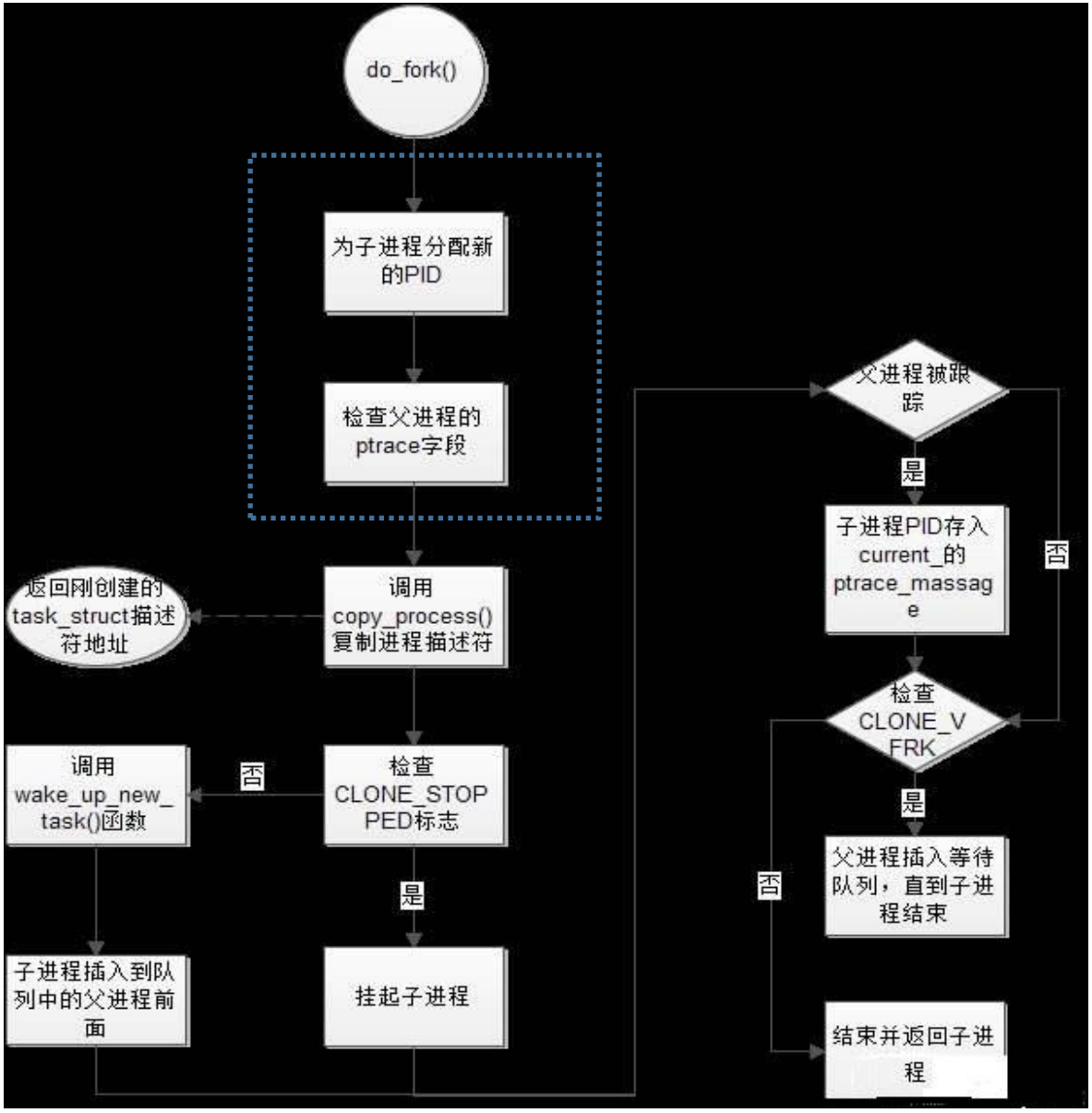
- clone通常用来创建用户线程
- clone是fork的升级版本，不仅可以创建进程或者线程，还可以指定创建新的命名空间（namespace）、有选择的继承父进程的内存、甚至可以将创建出来的进程变成父进程的兄弟进程等等。

```
#define ARCH_FORK() \
  INLINE_SYSCALL (clone, 4,                                          \
                  CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID | SIGCHLD, 0,     \
                  NULL, &THREAD_SELF->tid)
```

clone()参数标志

- CLONE_VM 表示在父子进程间共享 VM ;
- CLONE_FS 表示在父子进程间共享文件系统信息，包括工作目录等;
- CLONE_FILES 表示在父子进程间共享打开的文件;
- CLONE_SIGHAND 表示在父子进程间共享信号的处理函数;
- CLONE_PTRACE 表示如果父进程被跟踪，子进程也被跟踪;
- CLONE_VFORK 在 vfork 的时候使用;
- CLONE_PARENT 表示和复制的进程有同样的父进程;
- CLONE_THREAD 表示同一个线程组;

do_fork(…)执行流程



/kernel/fork.c

copy_process（…）

主要用来创建子进程的描述符以及与子进程相关数据结构.

1、检查clone_flags参数

2、dup_task_struct

3、检查资源限制

4、完成一些初始化工作

5、sched_fork：完成调度相关的设置，将这个task分配给CPU

6、复制共享进程的的各个部分，如页表等

7、分配进程PID并设置各个ID以及进程关系，等

/kernel/fork.c

内核线程创建

- 内核线程创建API
  - kthread_create
  - kthread_run

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

```
#define kthread_run(threadfn, data, namefmt, ...)                          \
({                                                                         \
        struct task_struct *__k                                            \
                = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
        if (!IS_ERR(__k))                                                  \
                wake_up_process(__k);                                      \
        __k;                                                               \
})
```

最终也是调用do_fork 函数实现

进程的终止

- 进程主动终止主要有如下两个途径：
  - ✓ 从main()函数返回，链接程序会自动添加对exit()系统调用。
  - ✓ 主动调用exit()系统调用。
- 进程被动终止主要有如下三个途径：
  - ✓ 进程收到一个自己不能处理的信号。
  - ✓ 进程在内核态执行的时候产生了一个异常。
  - ✓ 进程收到SIGKILL等终止信号
- exit()系统调用把退出码转换成内核要求的格式并且调用do_exit()函数来处理

0号进程&1号进程

- 进程0是指的的Linux内核初始化阶段从无到有创建的一个内核线程，称为
  - ✓ 进程0
  - ✓ idle进程
  - ✓ swapper进程

- Linux内核初始化函数start_kernel()在初始化完内核所需要的所有数据结构之后会创建另外一个内核线程，这个内核线程就是进程1或者叫做init进程

0号进程初始化

```
/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1fffff (=2MB)
 */
#define INIT_TASK(tsk)  \
{                                                                       \
        INIT_TASK_TI(tsk)                                               \
        .state          = 0,                                            \
        .stack          = init_stack,                                   \
        .usage          = ATOMIC_INIT(2),                               \
        .flags          = PF_KTHREAD,                                   \
        .prio           = MAX_PRIO-20,                                  \
        .static_prio    = MAX_PRIO-20,                                  \
        .normal_prio    = MAX_PRIO-20,                                  \
        .policy         = SCHED_NORMAL,                                 \
        .cpus_allowed   = CPU_MASK_ALL,                                 \
        .nr_cpus_allowed= NR_CPUS,                                      \
        .mm             = NULL,                                         \
        .active_mm      = &init_mm,                                     \
        .restart_block = {                                              \
                .fn = do_no_restart_syscall,                            \
        },                                                              \
        .se             = {                                             \
                .group_node     = LIST_HEAD_INIT(tsk.se.group_node),    \
        },                                                              \
        .rt             = {                                             \
                .run_list       = LIST_HEAD_INIT(tsk.rt.run_list),      \
                .time_slice     = RR_TIMESLICE,                         \
        },                                                              \
        .tasks          = LIST_HEAD_INIT(tsk.tasks),                    \
        INIT_PUSHABLE_TASKS(tsk)                                        \
        INIT_CGROUP_SCHED(tsk)                                          \
        .ptraced        = LIST_HEAD_INIT(tsk.ptraced),                  \
```

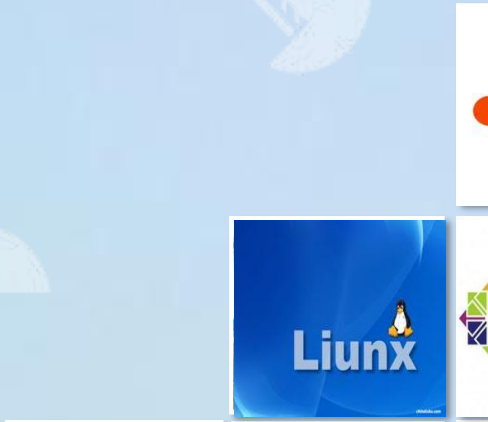
include/linux/init_task.h

僵尸进程&托孤进程

- 僵尸进程
  - ✓ 当一个进程通过exit()系统调用已经终止之后，进程处于僵尸状态（ZOMBIE）。
  - ✓ 当父进程通过调用wait()系统调用来获取已终结的子进程的信息之后，内核才会去释放子进程的task_struct数据结构

- 托孤进程
  - ✓ 如果父进程先于子进程消亡，那么子进程就变成孤儿进程

进程调度



进程分类

- CPU消耗型和IO消耗型
  - ✓ CPU消耗型：CPU-Bound。大部分时间都用在执行代码上，一直占用CPU。
  - ✓ IO消耗型：IO-Bound。大部分时间用来提交IO请求或者等待IO请求，很少占用CPU。
  - ✓ 通常IO消耗型的进程优先级要高于CPU消耗型。

第二种进程分类

- 批处理进程（batch processes）
  - ✓ 不必与用户交互，通常在后台运行
  - ✓ 不必很快响应
  - ✓ 典型的批处理程序：编译程序、科学计算...

- 交互式进程（interactive processes）
  - ✓ 需要经常与用户交互，因此要花很多时间等待用户输入操作
  - ✓ 响应时间要快，平均延迟要低于50~150ms
    典型的交互式程序：shell、文本编辑程序、图形应用程序等...

第二种进程分类

- 实时进程（real-time processes）
  - ✓ 有实时需求，不应被低优先级的进程阻塞
  - ✓ 响应时间要短、要稳定
  - ✓ 典型的实时进程：视频/音频、机械控制等

task_struct中关于优先级的成员

- static_prio是静态优先级，在进程启动时分配
- normal_prio是基于static_prio和调度策略计算出来的优先级，在创建进程时会继承父进程的normal_prio
- prio保存着进程的动态优先级，是调度类考虑的优先级，有些情况下需要暂时提高进程优先级，例如实时互斥量等
- rt_priority是实时进程的优先级。

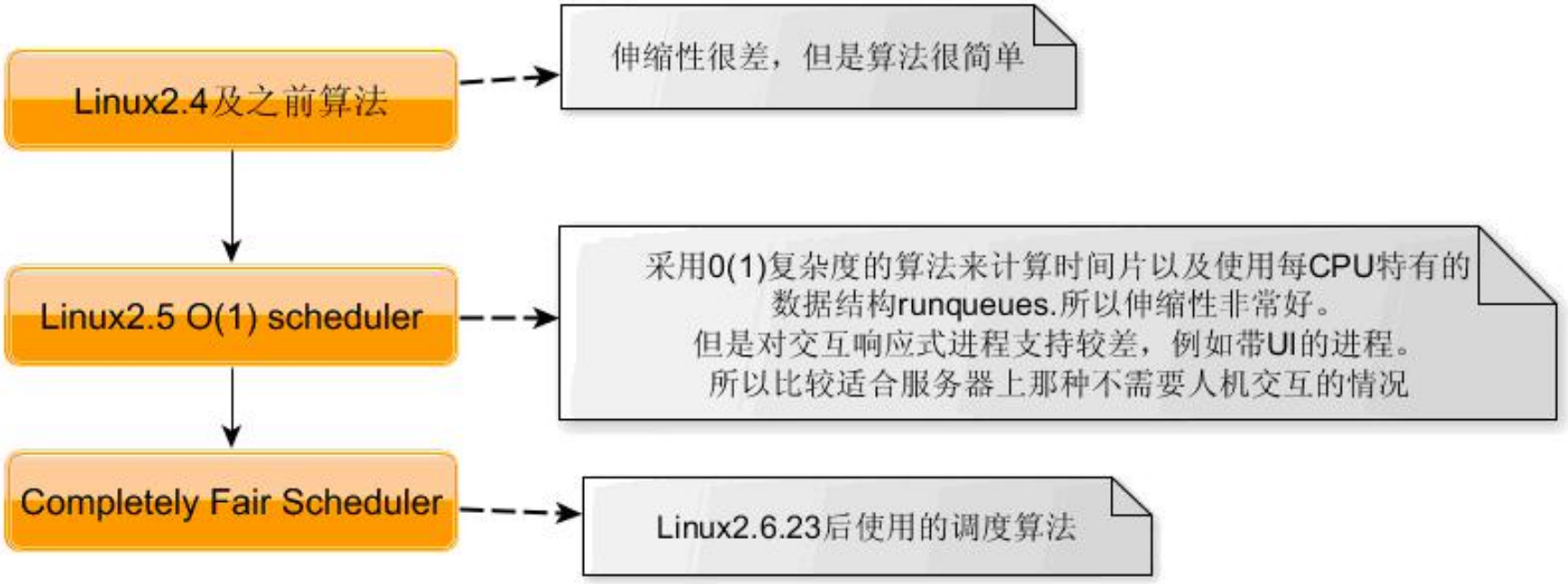
调度策略

- 非实时调度策略：
  - ✓ SCHED_NORMAL: 普通进程调度策略，使用CFS调度器来调度运行。
  - ✓ SCHED_BATCH:普通进程调度策略，使用CFS调度器。采用分时策略，根据动态优先级(可用nice()API设置），分配CPU运算资源。不会经常被抢占，牺牲交互交互性换取效率，适用于批处理任务。
  - ✓ SCHED_IDLE:普通进程调度策略，使用CFS调度器。适用于优先级较低的后台任务。

调度策略

- 实时调度策略
  - ✓ SCHED_FIFO: 实时进程调度策略，先入先出调度算法。相同优先级的任务先到先服务，高优先级的任务可以抢占低优先级的任务。
  - ✓ SCHED_RR: 实时进程调度策略，采用时间片，相同优先级的任务当用完时间片会被放到队列尾部，以保证公平性，同样，高优先级的任务可以抢占低优先级的任务。

Linux调度算法演变



Linux0.11内核调度算法

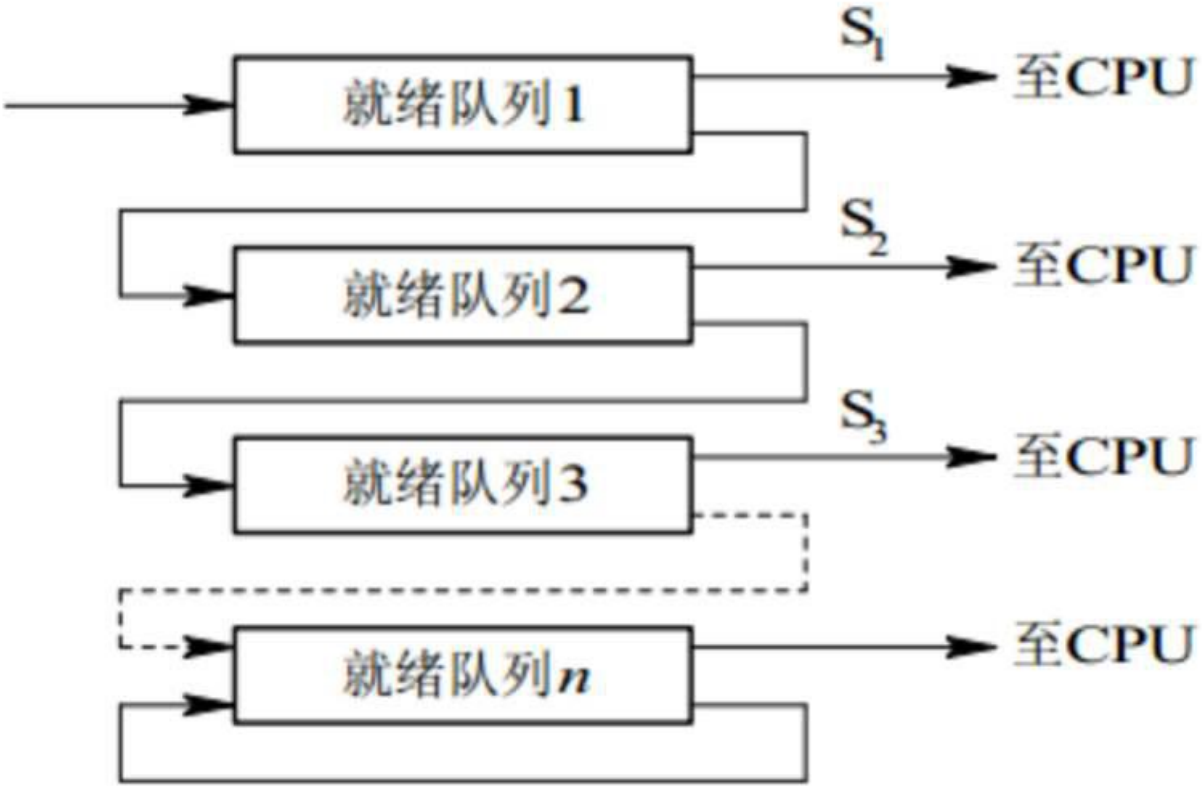
```
/* this is the scheduler proper: */

    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    switch_to(next);
```

遍历任务找到可运行的最大未使用时间片进程

O(n)调度器缺点

- Linux 0.11 ~ Linux 2.4实现的调度器算法是O(n)
- O(n)：选择下一个进程时间复杂度为O(n)
- 可扩展性差

多级反馈队列算法



(时间片：$S_1<S_2<S_3$)

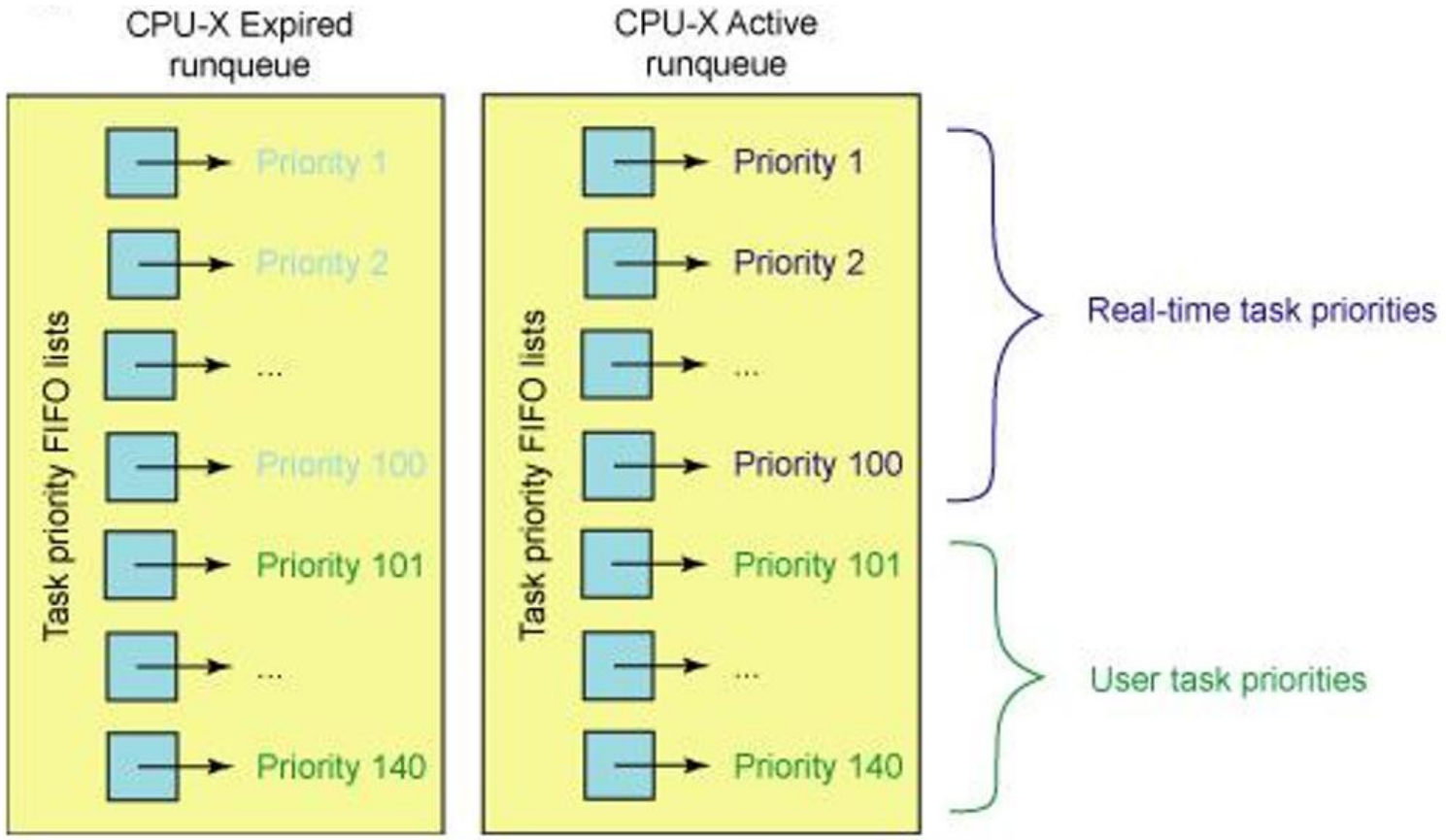
时间片 小 大

优先级 高 低

多级反馈队列算法的精髓

- 规则1：如果进程A的优先级大于进程B的优先级，那么调度器选择进程A。
- 规则2：如果进程A和进程B优先级一样，那么它们同属一个队列里，使用轮转调度算法来选择。
- 规则3：当一个新进程进入调度器时，把它放入到最高优先级的队列里。
- 规则4：当一个进程吃满了时间片，说明这是一个CPU消耗型的进程，那么需要把优先级降一级，从高优先级队列中迁移到低一级的队列里。
- 规则5：当一个进程在时间片还没有结束之前放弃CPU，那说明是一个IO消耗型的进程，那么优先级保持不变，维持原来的高优先级。

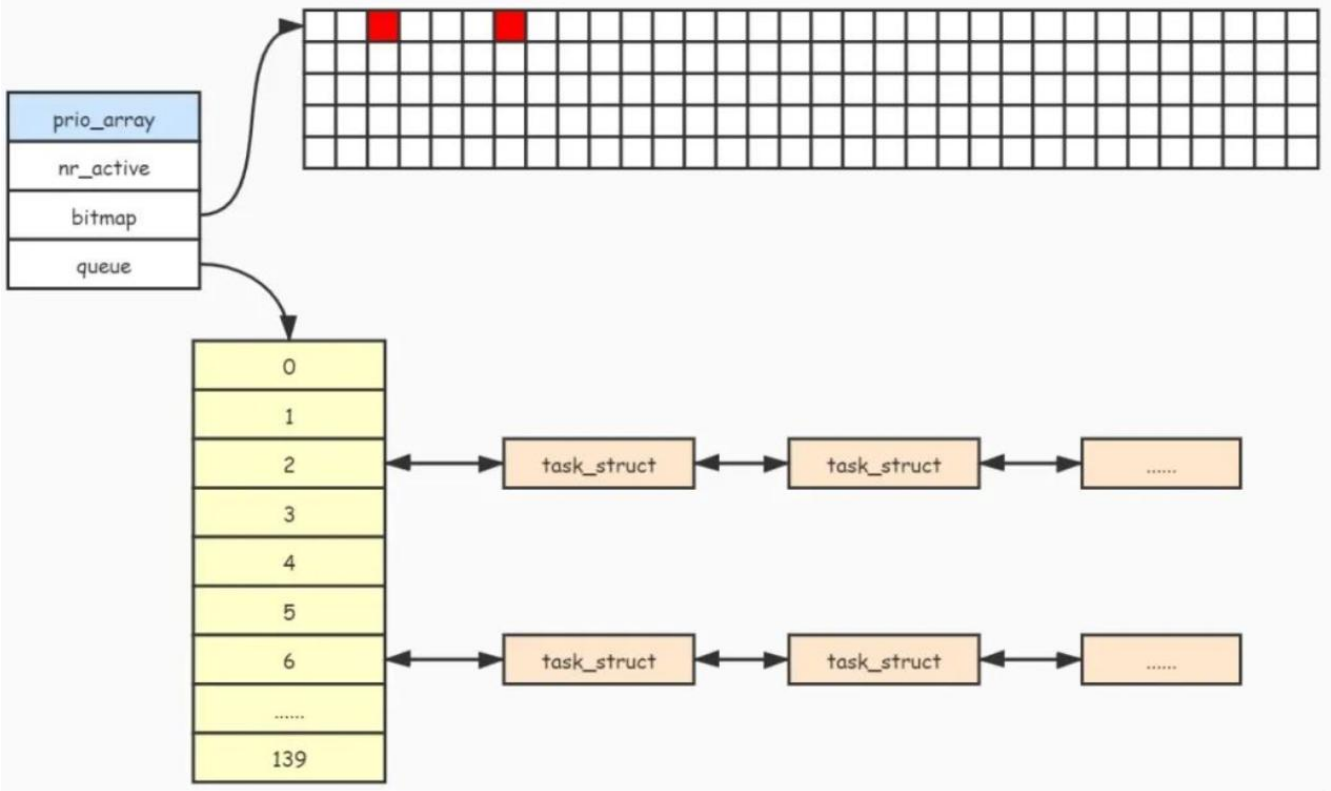
O(1)调度算法


CPU-X Expired runqueue
CPU-X Active runqueue
Task priority FIFO lists
Task priority FIFO lists
Priority 1
Priority 2
...
Priority 100
Priority 101
...
Priority 140
Priority 1
Priority 2
...
Priority 100
Priority 101
...
Priority 140
Real-time task priorities
User task priorities

O(1)调度算法

- 调度算法的核心思想是基于多级反馈队列算法。
- 每个CPU有一个就绪队列和一个到期队列，如果进程未用完其时间片则其在就绪队列，否则在过期队列。
- 就绪队列和到期队列有多个链表组成，每个优先级一个链表（前100个优先级用于实时进程，后40个用于用户普通进程）。
- 就绪队列为空时会与到期队列交换指针。

时间复杂度分析



- 若队列不为空则位图相应位染色，取最高优先级进程问题就简化为寻找一个 bitarray 里面最高位是 1 的 bit，时间复杂度为O(1)。

CFS调度算法

- CFS（Completely Fair Schedule），完全公平调度算法
- 调度思想：进程调度的效果应如同系统具备一个理想中的完美多任务处理器，每个进程都将获得1/n的处理器时间（n为可运行进程数量）。
- CFS允许每个进程运行一段时间、循环轮转，并选择运行虚拟时间最少的进程作为下一个运行进程。
- Nice值在CFS中被作为进程获得的处理器运行时间的权重，而不是用来计算具体的时间片。

CFS调度算法

- 时间记账：维护每个进程运行的记账时间，确保每个进程只在公平分配给它的处理器时间内运行
  - ✓ Vruntime变量存放进程的虚拟运行时间
- vruntime计算公式
  - ✓ vruntime = 实际运行时间 * nice为0的权重值/当前进程权重值
  - ✓ 每个调度实体的vruntime增加速度不同，权重越大的增加的越慢

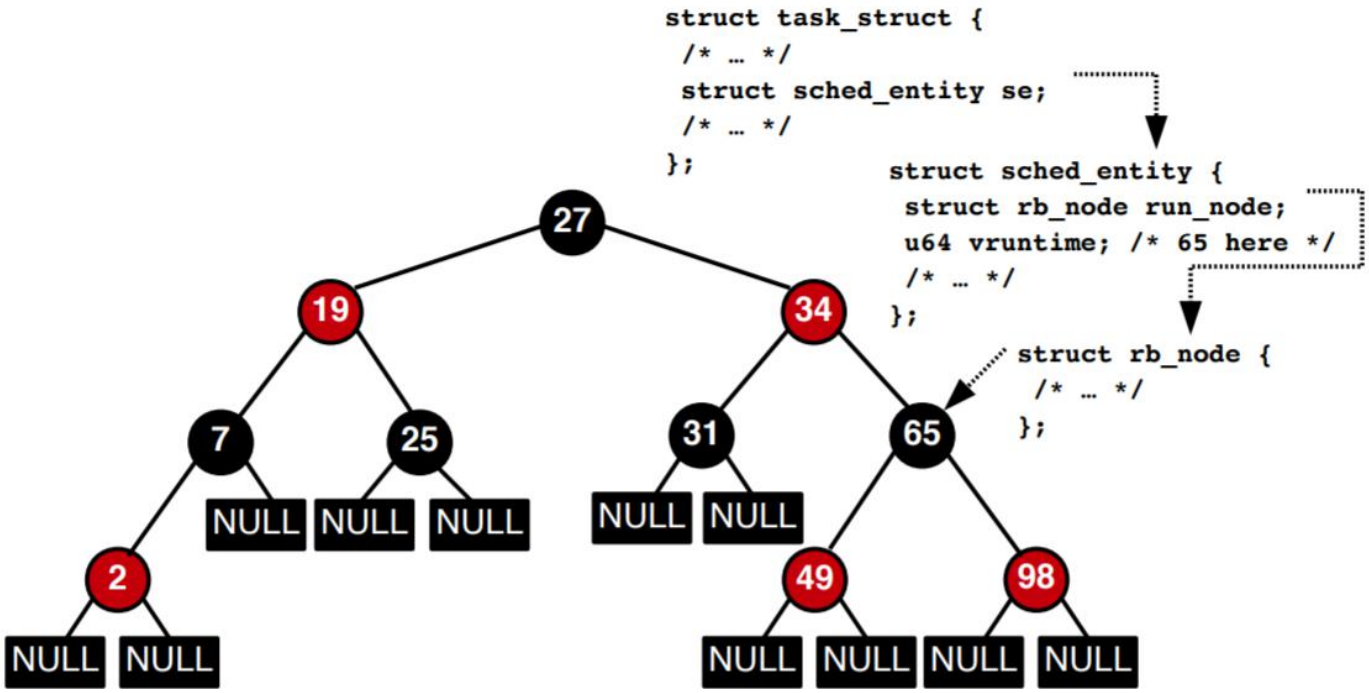
CFS调度算法

权重表：

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

CFS调度算法

- 进程选择
  - ✓ CFS维护一个以vruntime为索引的红黑树
  - ✓ 始终选择树中最左侧叶子节点为待运行任务



多核机器中的CFS调度

- 每个CPU都有一个运行队列
  - ✓ 避免了对共享数据的竞争访问
- 运行队列需保持平衡
  - ✓ 例如，双核运行队列中有一个高优先级进程的长运行队列，和一个低优先级进程的短运行队列，高优先级进程比低优先级进程获得更少的CPU时间
  - ✓ 需要根据进程优先级和CPU占用情况做负载均衡

调度时机

- 调度时机：
  - ✓ 阻塞操作：互斥量（mutex）、信号量（semaphore）、等待队列（waitqueue）等。
  - ✓ 在中断返回前和系统调用返回用户空间时，去检查TIF_NEED_RESCHED标志位以判断是否需要调度。
  - ✓ 将要被唤醒的进程（Wakeups）。
- 唤醒进程什么时候被调度？
  - ✓ 内核支持抢占
  - ✓ 内核不支持抢占

抢占和上下文切换

- 上下文切换是将当前在CPU上运行的进程交换到另一个进程的操作
- 由context_switch()执行，该函数由schedule()调用
  - ✓ switch_mm()：切换地址空间
  - ✓ switch_to()：切换CPU状态(寄存器)

抢占和上下文切换

- schedule()何时被调用?
  - ✓ 一个任务可以通过调用schedule()自动放弃CPU
  - ✓ 当前任务需要被抢占
    - 运行的时间足够长
    - 具有更高优先级的任务被唤醒

抢占和上下文切换

- need_resched标志
  - ✓ 表示是否应被重新调度
- need_resched何时被设置
  - ✓ scheduler_tick():当前运行的任务需要被抢
  - ✓ try_to_wake_up():唤醒具有更高优先级的进程

抢占和上下文切换

- need_resched标志在以下两种情况被检查
  - ✓ 返回用户空间(从系统调用或中断)
  - ✓ 从中断返回时
- 如果设置了标记，则调用schedule()

内核抢占

- Linux支持内核抢占
  - ✓ 只要执行处于安全状态，不持有任何锁，任务就可以在内核中被抢占
- thread_info结构中的preempt_count表示当前锁的深度，每当使用锁的时候该数值加1，释放锁数值减1
- 如果need_resched && !preempt_count，则表示可以被安全抢占

内核抢占

- 内核抢占发生在以下几种情况
  - ✓ 中断处理程序正在执行，且返回内核空间前
  - ✓ 内核代码再一次具有可抢占性的时候
  - ✓ 内核任务显式调用schedule()
  - ✓ 内核中任务阻塞（导致调用schedule）

进程同步和锁机制

临界资源和临界区

- 临界资源：一次仅允许一个进程访问的资源（硬、软）
- 临界区：各进程访问临界资源的代码

原子操作

- 原子操作可以保证指令以原子的方式执行，确保指令执行期间不被打断，要么全部执行完，要么根本不执行。
- 两个原子操作绝对不可能并发地访问同一个变量，也就绝不可能引起竞争。
- 内核提供了两组原子操作接口：
  - ✓ 一组针对整数进行操作
  - ✓ 一组针对单独的位进行操作

原子操作

```
typedef struct  {
    volatile int counter;
 }  atomic_t;
```

volatile 的特性

保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

禁止进行指令重排序。

自旋锁

- 自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个被争用的自旋锁，那么该线程就会一直进行忙循环－旋转－等待锁重新可用。
- 自旋锁可防止多于一个的执行线程同时进入临界区。
- 一个被争用的自旋锁使得请求它的线程在等待锁重新可用时自旋（特别浪费处理器时间），所以自旋锁不应该被长时间持有。
- 自旋锁可以在中断上下文中使用。

自旋锁基本使用形式

```
typedef struct {
volatile unsigned int slock;
 #ifdef CONFIG_DEBUG_SPINLOCK
  unsigned magic;
 #endif
  #ifdef CONFIG_PREEMPT
  unsigned int break_lock;
#endif
} spinlock_t;
```

- 因为自旋锁在同一时刻最多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区内。
- 自旋锁是不可递归的

在中断处理程序中使用自旋锁

- 在中断处理程序中使用自旋锁时，一定**要在获取自旋锁之前，首先禁止本地中断**。否则，中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已经被持有的自旋锁。
- 这样一来，中断处理程序就会自旋，等待该锁重新可用，但是锁的持有者在这个中断处理程序执行完毕前不可能运行。

→ 导致死锁

信号量

- Linux中的信号量是一种**睡眠锁**。
- 如果有一个任务试图获得一个已经被占用的信号量时，信号量会将其推进一个等待队列，然后让其睡眠。这时处理器可以去执行其它代码。
- 当持有信号量的进程将信号量释放后，处于等待队列中的那个任务将被唤醒，并获得该信号量。

```
struct semaphore
 {
atomic_t count;
 int sleepers;
wait_queue_head_t wait;
}
```

信号量特性

- 信号量可以同时允许任意数量的锁持有者，持有者的数量可以在声明信号量时指定，这个值称为使用者数量或叫数量 (count)，可分为：
  - ✓ 互斥信号量（二值信号量）：初值为1，在一个时刻仅允许有一个锁持有者。
  - ✓ 计数信号量：初值大于1，允许在一个时刻至多有count个锁持有者。
- 在内核中使用信号量时，基本用到的都是互斥信号量。

信号量操作

- 信号量支持两个原子操作：P( )和V( )，后来的系统把这两种操作分别叫做down( )和up( )，Linux也遵从这种叫法。
- down( )操作通过对信号量计数减1来请求获得一个信号量。
  - ✓ 如果结果≥0，获得信号量锁，任务进入临界区。
  - ✓ 如果结果<0，将任务放入等待队列。
- up( )操作用来释放信号量。如果在该信号量上的等待队列不为空，那么处于等待队列中的任务在被唤醒的同时会获得该信号量。

互斥体（Mutex）

- Mutex相当于计数为1的信号量
- 为什么需要有mutex这样一个互斥体？
  - ✓ Mutex的语义相对于信号量要简单轻便一些，在锁争用激烈的测试场景下，Mutex比信号量执行速度更快，可扩展性更好

互斥体（Mutex）

```
struct mutex {
    atomic_t                count;
     spinlock_t              wait_lock;
      struct list_head        wait_list;
    #ifdef CONFIG_DEBUG_MUTEXES
       struct thread_info      *owner;
       const char              *name;
        void                   *magic;
    #endif
    #ifdef CONFIG_DEBUG_LOCK_ALLOC
       struct lockdep_map      dep_map;
 #endif
     };
```

读写锁

- 信号量有一个明显的缺点—没有区分临界区的读写属性
- 读写锁通
- 常允许多个线程并发地读访问临界区，但是写访问只限制于一个线程。
- 特点：
  - ✓ 允许多个读者同时进入临界区，但同一时刻写者不能进入。
  - ✓ 同一时刻只允许一个写者进入临界区。
  - ✓ 读者和写者不能同时进入临界区

读写锁

```
typedef struct {
volatile unsigned int lock;
#ifdef CONFIG_DEBUG_SPINLOCK
 unsigned magic;
#endif
#ifdef CONFIG_PREEMPT
*/ unsigned int break_lock;
#endif } rwlock_t;
```

内存屏障

- 编译器可以:
  - ✓ 重新排序代码，只要它正确地维护了函数内和被调用函数的数据流依赖关系
  - ✓ 为了优化性能重新排序代码的执行
- 处理器可以:
  - ✓ 重新排序指令执行，只要它正确地维护了寄存器流的依赖关系
  - ✓ 只要内存修改正确地维护了数据流依赖关系，就对其重新排序
  - ✓ 重新排序指令的执行(用于性能优化)

内存屏障

- 内存屏障用于防止处理器或编译器对指令执行和内存修改进行重新排序。
- Load屏障：强制所有在load屏障指令之后的load指令，都在该load屏障指令执行之后被执行，并且一直等到load缓冲区被该CPU读完才能执行之后的load指令。
- Store屏障：强制所有在store屏障指令之前的store指令，都在该store屏障指令执行之前被执行，并把store缓冲区的数据都刷到CPU缓存。
- Full屏障：保障了早于屏障的内存读写操作的结果提交到内存之后，再执行晚于屏障的读写操作。

内存屏障函数接口

- rmb()：阻止跨越屏障的载入动作发生重排序
- read_barrier_depends()：阻止跨越屏障的具有数据依赖关系的载入动作重排序
- wmb()：阻止跨越屏障的存储动作发生重排序
- mb()：阻止跨越屏障的载入和存储动作重新排序
- smp_rmb()：在SMP上提供rmb()功能，在UP上提供barrier()功能
- smp_read_barrier_depends()：在SMP上提供read_barrier_depends()功能，在UP上提供barrier()功能
- smp_wmb()：在SMP上提供wmb()功能，在UP上提供barrier()功能
- smp_mb()：在SMP上提供mb()功能，在UP上提供barrier()功能
- barrier()：阻止编译器跨越屏障对载入或存储操作进行优化

Linux沙箱机制

沙箱机制——SELinux

自主访问控制（DAC）

- ✓ 对象（一般指文件等）的属主全权管理该对象的访问控制策略，有权读取、修改、转移对象信息，并且可以把这种权限转移给其他主体。

缺陷

- ✓ root 用户不受任何管制，系统上任何资源都可以无限制地访问
- ✓ 资源管理过于分散，给维护系统安全造成很大不便

沙箱机制——SELinux

强制访问控制（MAC）

- ✓ 每一个主体（包括用户和程序）和客体都拥有固定的安全标记，主体能否对客体进行相关操作，取决于主体和客体所拥有安全标记的关系（如：安全标记同为“机密”则可以执行操作）

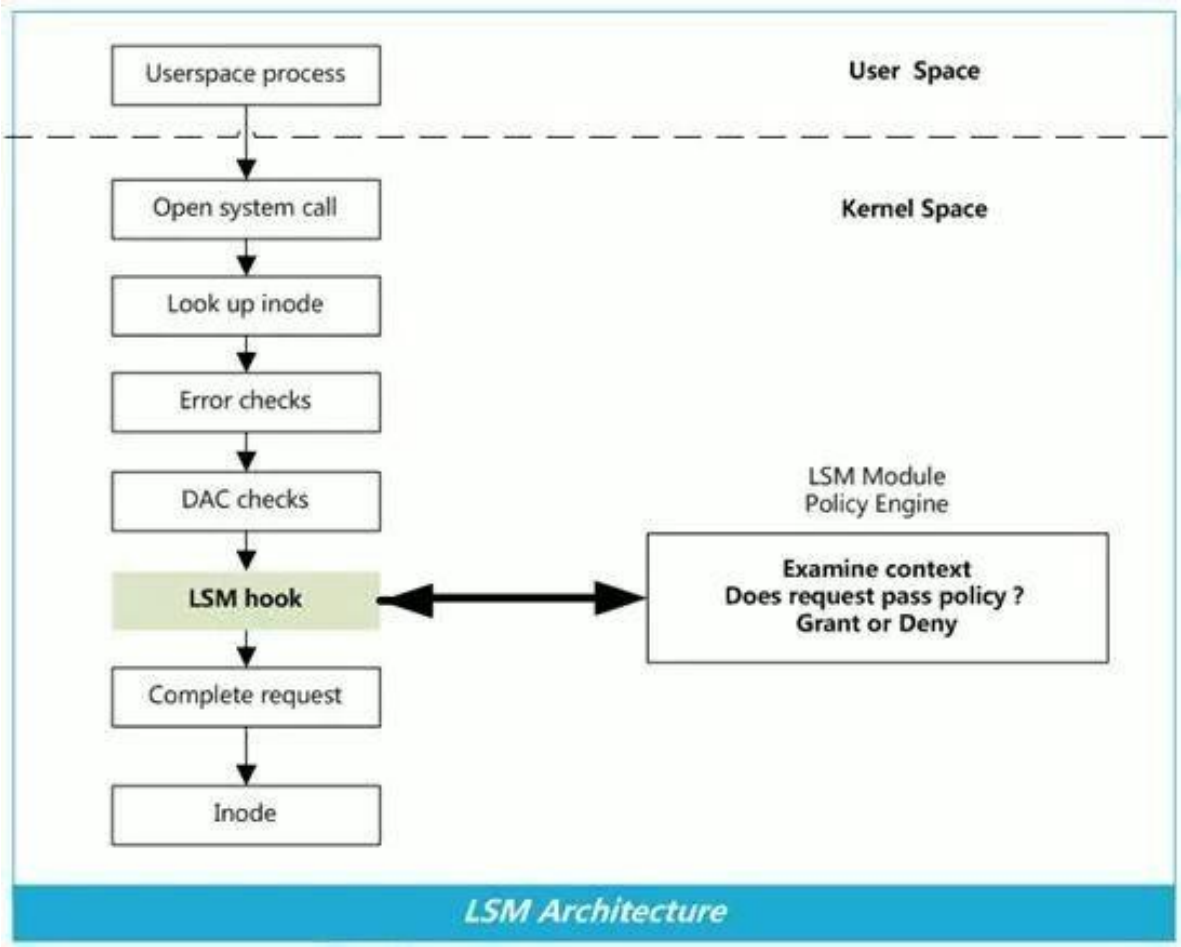
沙箱机制——SELinux

SELinux三种工作模式：

- ✓ 强制模式（违反规则被阻止并记录日志）
- ✓ 容忍模式（违反规则不阻止只记录日志）
- ✓ 关闭模式（不生效）

沙箱机制——SELinux

- ✓ SELinux在Linux内核以安全模块LSM（Linux Security Modules）的形式集成进来



沙箱机制——chroot

- chroot用于更改根目录
  - ✓ 最初用来测试系统代码安全性
  - ✓ 将进程对文件的访问限制在一个指定的目录中
  - ✓ 在新根下将访问不到旧系统的根目录结构和文件，增强系统的安全性
  - ✓ 注意在chroot()调用之前使用chdir()调用

沙箱机制——chroot

- 逃逸示例

```
mkdir(/temp)          /* create temp directory                */
chroot(/temp)         /* now current dir is outside jail      */
chdir(" ../../../.")  /* move current dir to true root dir    */
chroot(".")           /* out of jail                          */
```

沙箱机制——Seccomp

- Seccomp(secure computing mode)
  - ✓ 将进程可用的系统调用限制为四种：read，write，_exit，sigreturn（白名单）
  - ✓ 2.6.12版本(2005年3月8日)中引入linux内核
- Seccomp-BPF
  - ✓ Seccomp-BPF是Seccomp和BPF规则的结合，它允许用户使用可配置的策略过滤系统调用，该策略使用Berkeley Packet Filter规则实现，可以对任意系统调用及其参数）进行过滤

沙箱机制——Seccomp

- 示例

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/prctl.h>
#include <linux/seccomp.h>

int main() {
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
    char *buf = "hello world!\n";
    write(0,buf,0xc);
    printf("%s",buf);
}
```

```
root@WANG:/opt# gcc seccomp.c -o out
root@WANG:/opt#
root@WANG:/opt# ./out
hello world!Killed
```

思考题

1、用户空间进程的页表是什么时候分配的？

2、fork和clone的区别？

3、do_fork函数的作用和主要功能？

4、什么是写时复制？Linux在进程创建时是怎样实现COW的？

5、CFS调度器是如何工作的？

6、进程沙箱的作用是什么？有哪些实现方式