中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核
分析与实践

3. 进程管理与调度

授课教师：游伟 副教授

授课时间：周五16:00 – 17:30（公教三楼3505）

上机时间：周五18:00 – 19:30（理工配楼208B机房）

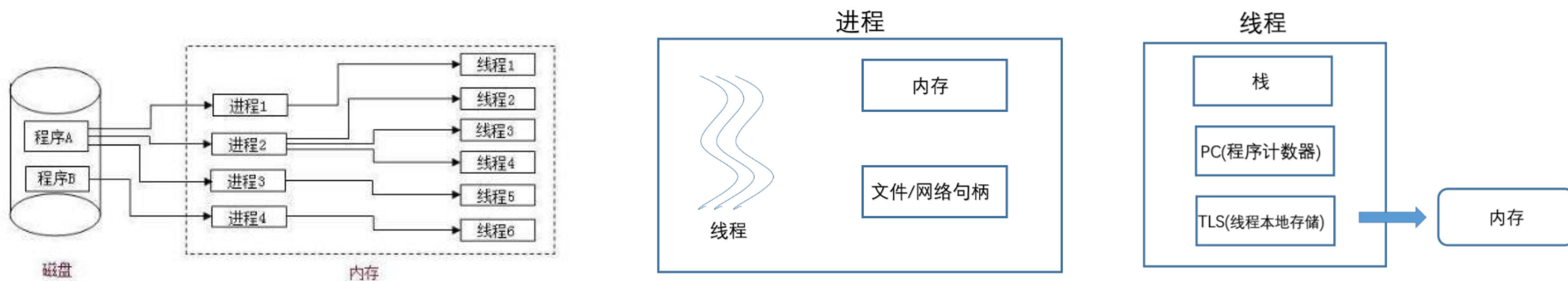
课程主页：https://www.youwei.site/course/kernel

目录

1. 基本概念与关键数据结构
2. 进程管理
3. 进程调度

3.1 基本概念与关键数据结构

- 进程与程序：
  - 程序是存储在磁盘上的一系列代码和数据；一个进程是一个正在执行程序的实例，包括程序计数器、变量的当前值、寄存器、输入输出、状态。
  - 进程是一次运行的活动，是一个动态概念；程序是一组有序的静态指令，是一个静态概念。进程是执行程序的动态过程；程序是进程运行的静态文本。
- 进程与线程
  - 进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础
  - 线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。



操作系统的进程控制块（PCB）

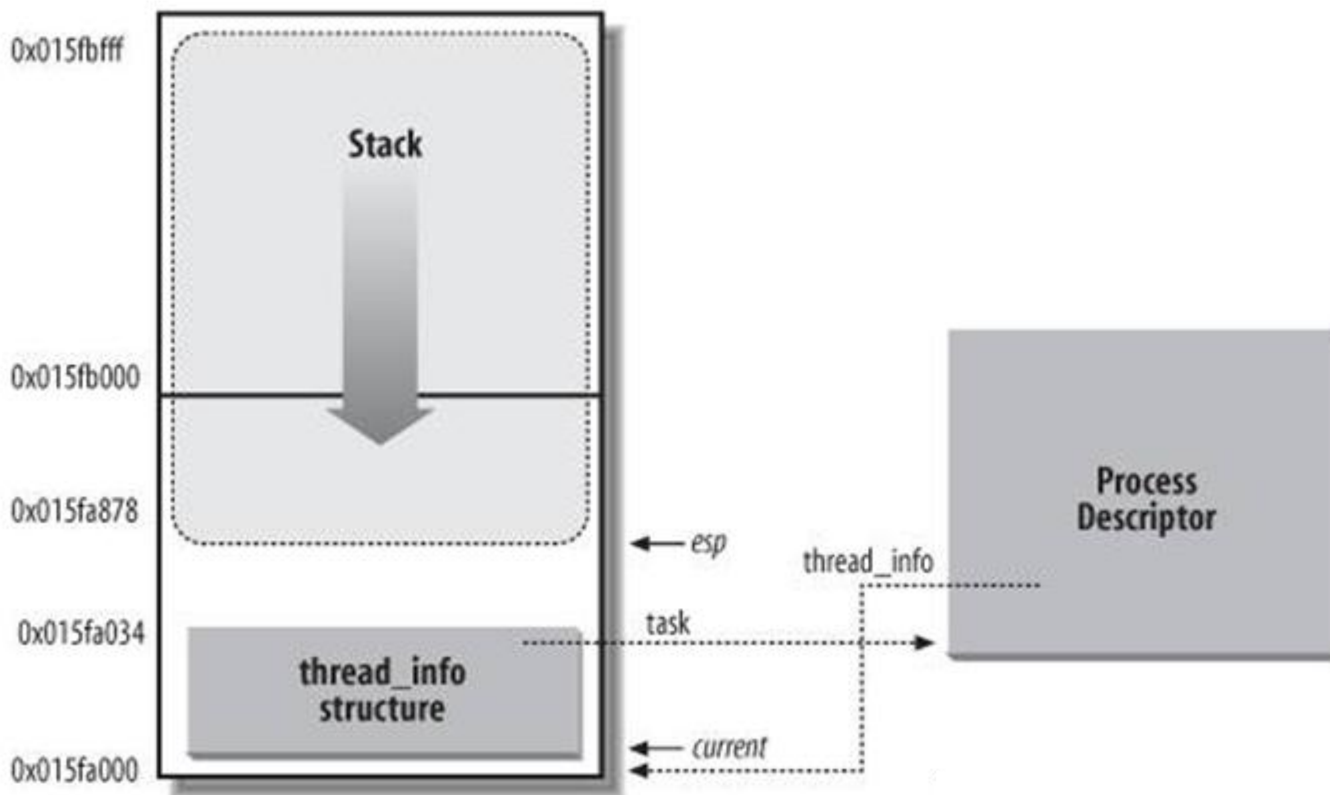
- 进程的运行状态：包括就绪、运行、等待阻塞等状态
- 程序计数器：记录当前进程运行到哪条指令了
- CPU寄存器：主要用于保存当前运行的上下文，记录CPU所有必须保存下来的寄存器信息，以便进程调度出去之后还能调度回来并接着运行
- CPU 调度信息：包括进程优先级、调度队列等相关信息
- 内存管理信息：进程使用的内存信息，如进程的页表等
- 文件相关信息：包括进程打开的文件等
- 统计信息：包含进程运行时间等相关的统计信息

task_struct & thread_info & stack

- 各自作用:
  - task_struct：进程描述符，记录所有进程信息（调度、文件、内存、信号等）。
  - thread_info：线程描述符，保存与体系结构相关的、上下文切换时需要快速访问的字段（如 flags、preempt_count、CPU info）。
  - stack：内核栈，每个线程都有独立的内核栈。
- 旧版本（Linux 4.x）
  - thread_info 在内核栈的最低地址（栈底）。内核栈是固定大小（例如 8KB 或 16KB），分配的时候把 thread_info 放在最低端，剩下部分用作栈。
  - 通过当前栈指针（sp）掩码操作，就能快速算出 thread_info 的地址。
- 新版本（>= Linux 5.10+）
  - 后来内核社区觉得：thread_info 太小，不值得占用一页栈底空间；更关键的是 内核栈随机化 / 安全性 以及 可变大小栈 需要把 thread_info 移出栈底。
  - 因此thread_info 不再放在内核栈底，而是内嵌在 task_struct 里。

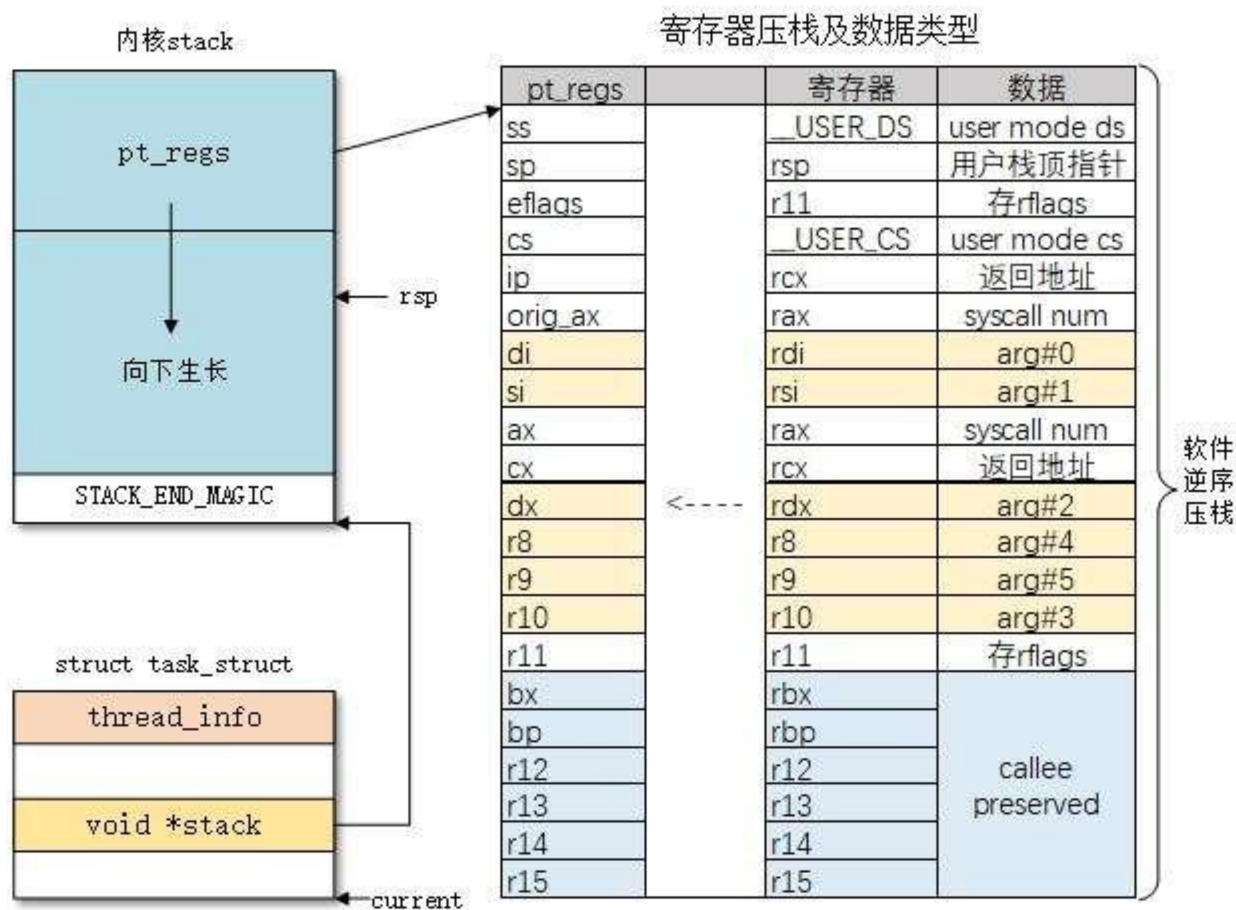
task_struct & thread_info & stack

- 旧版本内核三者布局



task_struct & thread_info & stack

- 新版本内核三者布局



如何获得当前进程的task_struct

- 内核使用一个 per-CPU 变量 current_task 保存当前 CPU 正在运行的进程指针。
  - 在 进程上下文切换 时，内核调用 switch_to(prev, next)。
  - 切换过程中，内核会把即将运行的进程 next 的 task_struct * 写入 per-CPU 变量 current_task。
  - 这样每个 CPU 的 current_task 始终指向该 CPU 上正在执行的进程。
- 获取当前进程时，通过 this_cpu_read_stable(current_task) 直接读出指针。

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
        return this_cpu_read_stable(current_task);
}

#define current get_current()
```

Linux内核的进程描述符（task_struct）

/ include / linux / sched.h   All symbols   Search Identifier

```
591
592 struct task_struct {
593 #ifdef CONFIG_THREAD_INFO_IN_TASK
594         /*
595          * For reasons of header soup (see current_thread_info()), this
596          * must be the first element of task_struct.
597          */
598         struct thread_info              thread_info;
599 #endif
600         /* -1 unrunnable, 0 runnable, >0 stopped: */
601         volatile long                   state;
602
603         /*
604          * This begins the randomizable portion of task_struct. Only
605          * scheduling-critical items should be added above here.
606          */
607         randomized_struct_fields_start
608
609         void                            *stack;
610         atomic_t                        usage;
611         /* Per task flags (PF_*), defined further below: */
612         unsigned int                    flags;
613         unsigned int                    ptrace;
614
615 #ifdef CONFIG_SMP
616         struct llist_node               wake_entry;
617         int                             on_cpu;
618 #ifdef CONFIG_THREAD_INFO_IN_TASK
619         /* Current CPU: */
620         unsigned int                    cpu;
621 #endif
622         unsigned int                    wakee_flips;
623         unsigned long                   wakee_flip_decay_ts;
624         struct task_struct              *last_wakee;
625
```

3.1.1 进程属性相关信息

- 进程属性相关信息主要包括和进程状态相关的信息，如进程状态、PID等信息。
- task_stuct 数据结构中关于进程属性的一些重要成员如下。
  - pid 成员：这是进程唯一的标识符(identifer)。pid_t的类型是整数类型，pid 默认的最大值见/proc/sys/kernel/pid_max 节点。
  - comm 成员:用于存放可执行程序的名称。
  - state 成员：用于记录进程的状态，进程的状态主要有 TASK_RUNNING、TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、EXIT_ZOMBIE、TASK_DEAD等。
  - real_cred 和 cred 成员:用于存放进程的一些认证信息。

```
struct cred {
        atomic_t        usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
        atomic_t        subscribers;    /* number of processes subscribe
        void            *put_addr;
        unsigned        magic;
#define CRED_MAGIC      0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
        kuid_t          uid;            /* real UID of the task */
        kgid_t          gid;            /* real GID of the task */
        kuid_t          suid;           /* saved UID of the task */
        kgid_t          sgid;           /* saved GID of the task */
        kuid_t          euid;           /* effective UID of the task */
        kgid_t          egid;           /* effective GID of the task */
        kuid_t          fsuid;          /* UID for VFS ops */
```

3.1.1 进程属性相关信息

- cred：指向当前进程正在使用的凭证（credentials），包括有效的 UID/GID、权限位、capabilities 等。内核在访问控制时（比如 open() 文件）用它来判断是否有权限。
- real_cred：指向进程的“真实身份”凭证，通常保存进程最初的 UID/GID（或者 setuid 前的 UID），用来标识进程真正的拥有者。
- 关系：
  - cred 可能会因为 setuid()、提权、临时降权而改变。
  - real_cred 一般保持不变，表示该进程的真实属主。
  - 二者可以相同，也可以不同（例如 setuid 程序中，real_cred 仍是调用者，而 cred 被修改为目标用户）。
- 为什么要分两个：
  - 保留一个稳定的“真实身份” (real_cred) 以便审计、信号权限检查。
  - 同时允许进程有一个“当前有效身份” (cred) 用于权限判断。
- 认证过程：
  - 当进程执行敏感操作（如文件访问、发送信号），内核会查 cred 来决定是否允许。
  - 当需要判断“是否是某用户真正拥有的进程”时（如 kill 里 UID 检查），会比较 real_cred。

3.1.1 进程属性相关信息

场景	程序类型	运行用户	real_cred->uid	real_cred->euid	cred->uid	cred->euid	说明
1	普通程序	user	1000	1000	1000	1000	real_cred 与 cred 相同,
2	root程序	user	1000	1000	1000	0	执行 root 程序，cred->euid 提权到 root，用于权限检查，其余信息保留
3	root程序	root	0	0	0	0	root 运行 root 程序，本身权限即最高，real_cred 与 cred 相同
4	临时降权程序	root 启动，调用 seteuid(1000)	0	0	0	1000	内核中 cred->euid 被修改为 1000，real_cred 保持原 root

3.1.2 进程间的关系

- 操作系统的第一个进程是空闲进程(或者叫作进程 0)。此后，每个进程都有一个创建它的父进程，进程本身也可以创建其他的进程，父进程可以创建多个进程。进程类似于一个家族，有父进程、子进程，还有兄弟进程。
- task_struct 数据结构中涉及进程间关系的重要成员如下。
  - real_parent：指向当前进程的真实父进程，也就是最初调用 fork() 创建它的那个进程。
  - parent：指向当前进程在内核中用于信号和退出处理的父进程。
    - 当父进程退出时，内核会把孤儿进程的 parent 改为 init（PID 1），以保证进程能被回收，但 real_parent 仍然保持原父进程不变。
  - 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。

3.1.3 进程调度相关信息

- 进程一个很重要的角色是作为一个调度实体参与操作系统中的调度，这样就可以实现CPU的虚拟化，即每个进程都感觉直接拥有了CPU。
- 从宏观上看，各个进程都是并行运行的，但是微观上看每个进程都是串行执行的。
- 进程调度是操作系统中的一个核心功能，这里先暂时列出Linux内核的task_struct数据结构中关于进程调度的一些重要成员。
  - prio 成员:保存着进程的动态优先级，这是调度类考虑的优先级。
  - static_prio 成员:静态优先级，在进程启动时分配。内核不仔储 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_prionity 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:用于确定进程的类型，比如是普通进程还是实时进程。
  - cpus_allowed 成员:用于确定进程可以在哪几个 CPU 上运行。

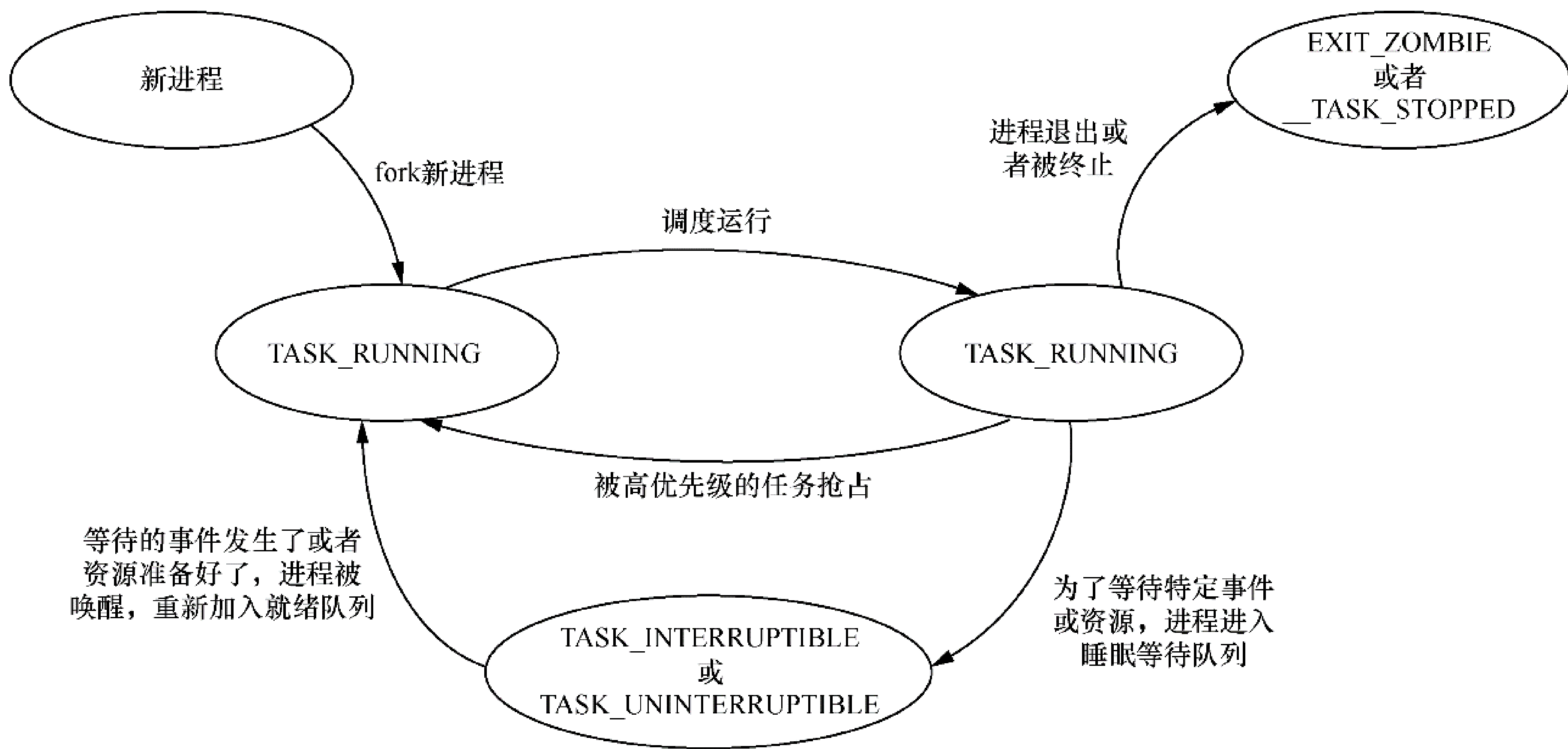
3.1.4 内存管理与文件管理相关信息

- 进程在运行之前需要先加载到内存，因此进程描述符里必须有抽象描述内存管理的相关信息，必须有一个指向 mm_struct 数据结构的指针 mm。
- 此外，进程在生命周期内总是需要通过打开文件、读写文件等操作来完成一些任务，这就和文件系统密切相关了。
- task_struct 数据结构中与内存管理和文件管理相关的重要成员如下。
  - mm 成员:指向进程所管理的内存中总的抽象数据结构 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。

3.2 进程管理

- 进程的生命周期
- 进程的关系
- 进程的创建与终结
- Linux中的线程

3.2.1 进程的生命周期



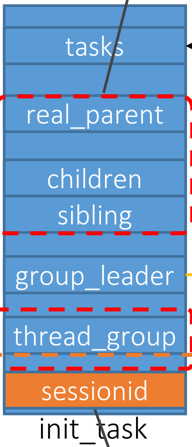
3.2.1 进程的生命周期

- TASK_RUNNING：可运行态，进程或许正在运行，或许在就绪队列中等待运行
- TASK_INTERRUPTIBLE：可中断睡眠态（浅睡眠状态），进程被阻塞等待某些条件的达成或者某些资源的就位，一旦条件达成或者资源就位，内核就可把进程的状态设置成 TASK_RUNNING，并将其加入就绪队列
- TASK_UNINTERRUPTIBLE：不可中断睡眠态（深度睡眠状态)，进程在睡眠等待时不受干扰，对信号不做反应，不可以发送 SIGKILL 信号使它们终止，因为它们不响应信号
- __TASK STOPPED：终止态，进程停止运行
- EXIT_ZOMBIE：僵尸态， 进程已经消亡，但是 task_struct 数据结构还没有释放，子进程退出时，父进程可以通过 wait() 或者 waitpid() 来获取子进程消亡的原因

进程生命周期实例

```c
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <fcntl.h>
5.  #include <sys/wait.h>
6.  #include <errno.h>

7.  int main() {
8.      pid_t pid;

9.      pid = fork();
10.     if (pid < 0) {
11.         perror("fork");
12.         exit(1);
13.     }
14.     if (pid == 0) {
15.         // Child process
16.         printf("[child] PID=%d, running...\n",
    getpid());
17.         // TASK_RUNNING: executing code before sleep
18.         sleep(2);

19.         int fd = open("/dev/sda", O_RDONLY);
20. if (fd < 0) {
21.             perror("[child] open");
22.         } else {
23.             char buf[1024];
24.             // TASK_UNINTERRUPTIBLE: simulate
    blocking I/O
```

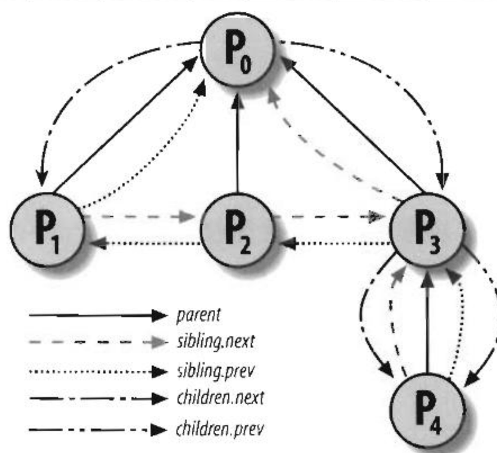
```c
1.                  printf("[child] blocking read -> should
    enter UNINTERRUPTIBLE (D)\n");
2.                  read(fd, buf, sizeof(buf));  // will
    block
3.                  close(fd);
4.              }
5.              printf("[child] exit -> will become ZOMBIE
    until parent waits\n");
6.              _exit(0);  // EXIT_ZOMBIE
7.      } else {
8.          // Parent process
9.          printf("[parent] PID=%d, child PID=%d\n",
    getpid(), pid);

10.         // TASK_INTERRUPTIBLE: waitpid() puts parent
    to sleep (S)
11.         printf("[parent] waiting for child
    (INTERRUPTIBLE)...\n");
12.         int status;
13.         waitpid(pid, &status, 0);
14.         // After wait, child ZOMBIE is reaped
15.         printf("[parent] child reaped, parent exits
    (RUNNING)\n");
16.     }
17.     return 0;
18. }
```



家族树
线程组
tasks
real_parent
children
sibling
group_leader
thread_group
sessionid
init_task
……
会话组1
会话组2

3.2.2 进程的关系

- 进程家族树
  - 内核启动时会创建一个 init_task 进程，称为进程0 或 idle 进程。当系统没有进程需要调度时，调度器就会运行 idle 进程
  - 系统初始化快完成时会创建一个 init 进程，这就是常说的进程 1，它是所有进程的祖先，从这个进程开始所有的进程都参与了调度
  - 若进程P_0创建了进程P_3，则进程P_0为父进程，进程P_3为子进程；若进程P_3创建了进程P_4，那么进程P_0和进程P_4之间的关系就是祖孙关系
  - 若进程P_0创建了 P_1, P_2, P_3 进程，这些 P_i $(1 \le i \le 3)$ 进程称为兄弟进程



```c
// File: chapter3/Relation/process.c

1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <unistd.h>

4.   int main() {
5.       pid_t p1, p2, p3, p4;
6.       printf("P0 (pid=%d, ppid=%d) started\n", getpid(), getppid());
7.       p1 = fork();
8.       if (p1 == 0) {
9.           printf("P1 (pid=%d, ppid=%d)\n", getpid(), getppid());
10.          pause();
11.          exit(0);
12.      }
13.      p2 = fork();
14.      if (p2 == 0) {
15.          printf("P2 (pid=%d, ppid=%d)\n", getpid(), getppid());
16.          pause();
17.          exit(0);
18.      }
19.      p3 = fork();
20.      if (p3 == 0) {
21.          printf("P3 (pid=%d, ppid=%d)\n", getpid(), getppid());
22.          p4 = fork();
23.          if (p4 == 0) {
24.              printf("P4 (pid=%d, ppid=%d)\n", getpid(), getppid());
25.              pause();
26.              exit(0);
27.          }
28.          pause();
29.          exit(0);
30.      }
31.      pause();
32.      return 0;
33.  }
```

```
❯ ./test
P0 (pid=3514979, ppid=3512176) started
P1 (pid=3514980, ppid=3514979)
P2 (pid=3514981, ppid=3514979)
P3 (pid=3514982, ppid=3514979)
P4 (pid=3514983, ppid=3514982)
```

```
    PID    PPID CMD
3514979 3512176 ./test
3514980 3514979  \_ ./test
3514981 3514979  \_ ./test
3514982 3514979  \_ ./test
3514983 3514982      \_ ./test
```

遍历进程家族树

```
1.   struct task_struct *task, *sib, *child;
2.   struct list_head *p;

3.   // 遍历系统中的所有进程
4.   for_each_process(task) { show_info(task); }
```

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

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

```
5.   // 遍历当前进程的所有祖先进程
6.   for (task = current; task != &init_task; task = task->parent) { show_info(task); }

7.   // 遍历当前进程的所有兄弟进程
8.   list_for_each_entry(sib, task->sibling.prev; sibling) { show_info(sib); }

9.   // 遍历当前进程的所有子进程
10.  list_for_each(p, &(task->children))
11.  {
12.    child = list_entry(p, struct task_struct, sibling);
13.    show_info(child);
14.  }
```

3.2.2 进程的关系

- 线程组
  - 一个线程组中的所有线程使用和该线程组中的主线程相同的标识符，即该组中第一个进程的 pid，它被存入 task_struct 数据结构的 tgid 成员中
  - 一个多线程应用程序中的所有线程必须有相同标识符，这样可以把指定信号发送给组里所有的线程
  - 一个进程创建后，若整个线程组只有这个进程，那么它的 pid 和 tgid 是一样的；当进程创建了一个新的线程之后，新线程拥有属于自己的 pid，但是它的 tgid 还是该进程的 tgid，因为它和该进程同属一个线程组
  - getpid() 系统调用返回当前进程的 tgid，而不是线程的 pid，因为一个多线程应用程序中的所有线程共享相同的标识符；gettid() 系统调用会返回线程的 pid

```c
// File: chapter3/Relation/thread.c

1.   #include <stdio.h>
2.   #include <pthread.h>
3.   #include <unistd.h>

4.   struct parameter { int thread_id; char *thread_name; };

5.   void * worker(void *arg)
6.   {
7.     int thread_id = ((struct parameter *)(arg))->thread_id;
8.     char *thread_name = ((struct parameter *)(arg))->thread_name;

9.     while (1);
10.  }

11.  int main()
12.  {
13.    pthread_t p1, p2, p3;
14.    struct parameter param1, param2, param3;
15.
16.    param1.thread_id = 1;
17.    param1.thread_name = "A";
18.    param2.thread_id = 2;
19.    param2.thread_name = "B";
20.    param3.thread_id = 3;
21.    param3.thread_name = "C";
22.
23.    pthread_create(&p1, NULL, worker, &param1);
24.    pthread_create(&p2, NULL, worker, &param2);
25.    pthread_create(&p3, NULL, worker, &param3);
26.
27.    pthread_join(p1, NULL);
28.    pthread_join(p2, NULL);
29.    pthread_join(p3, NULL);
30.  }
```

```
      tgid    pid
user  2401   2401  0.0  0.0  27236  616 pts/0  Sl+  06:11  0:00 ./thread   main
user  2401   2402 61.9  0.0  27236  616 pts/0  Rl+  06:11  0:19 ./thread   1A
user  2401   2403 61.1  0.0  27236  616 pts/0  Rl+  06:11  0:19 ./thread   2B
user  2401   2404 60.8  0.0  27236  616 pts/0  Rl+  06:11  0:19 ./thread   3C
```

遍历线程组

```
1.   struct task_struct *process, *thread;

2.   // method-1: iterate through the thread_node linked list
3.   //            whose head is in the signal_struct shared by the thread group
4.   for_each_process_thread(process, thread)
5.   {
6.       show_info(thread);
7.   }
```

```
#define __for_each_thread(signal, t)    \
    list_for_each_entry_rcu(t, &(signal)->thread_head, thread_node)

#define for_each_thread(p, t)       \
    __for_each_thread((p)->signal, t)

#define for_each_process_thread(p, t)   \
    for_each_process(p) for_each_thread(p, t)
```

```
8.   // method-2: iterate through the thread_group linked list
9.   //            whose head is in the task_struct of the thread group leader
10.  for_each_process(process)
11.  {
12.      show_info(process);
13.      list_for_each_entry(thread, &(process->thread_group), thread_group)
14.      {
15.          show_info(thread);
16.      }
17.  }
```

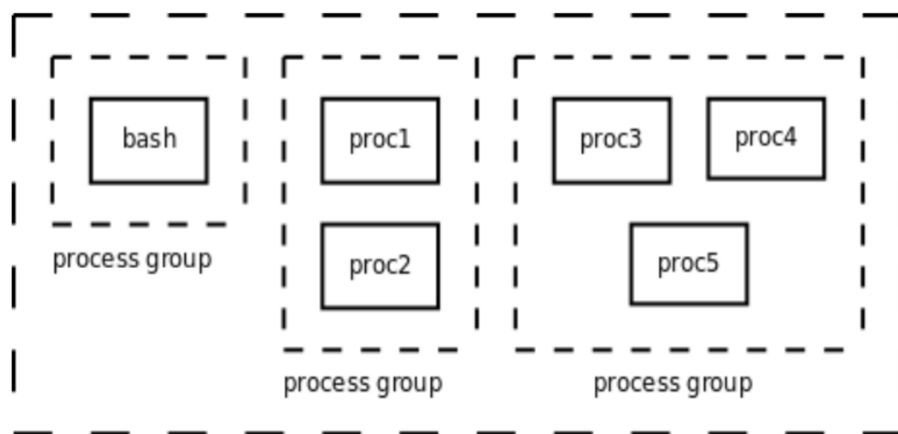
3.2.2 进程的关系

- 进程组与会话
  - 每个进程都属于一个进程组，每个进程组中可以包含多个进程。进程会有一个进程组领导，其 pid 成为识别进程组的 pgid
  - 多个进程组还可以构成一个会话，会话是由其中的进程建立的，该进程叫做会话的领导进程，其 pid 成为识别会话的 sid
  - 会话中的每个进程组称为一个工作 (job)。会话可以有一个进程组成为会话的前台工作 (foreground)，而其他的进程组是后台工作 (background)
  - 每个会话可以连接一个控制终端 (control terminal)。当控制终端有输入输出时，都传递给该会话的前台进程组；由终端产生的信号，比如 CTRL+Z， CTRL+\，会传递到前台进程组
  - 会话的意义在于将多个工作囊括在一个终端，并取其中的一个工作作为前台，来直接接收该终端的输入输出及终端信号，其他工作在后台运行

进程组与会话操作实例

```
$ proc1 | proc2 &
$ proc3 | proc4 | proc5
```

- shell 属于一个单独的进程组，proc1、proc2 属于同一个后台进程组，proc3、proc4、proc5 属于同一个前台进程组
- 这些进程组的控制终端相同，它们属于同一个session，当用户在控制终端输入特殊控制键时，内核会发送相应信号给前台进程组的所有进程



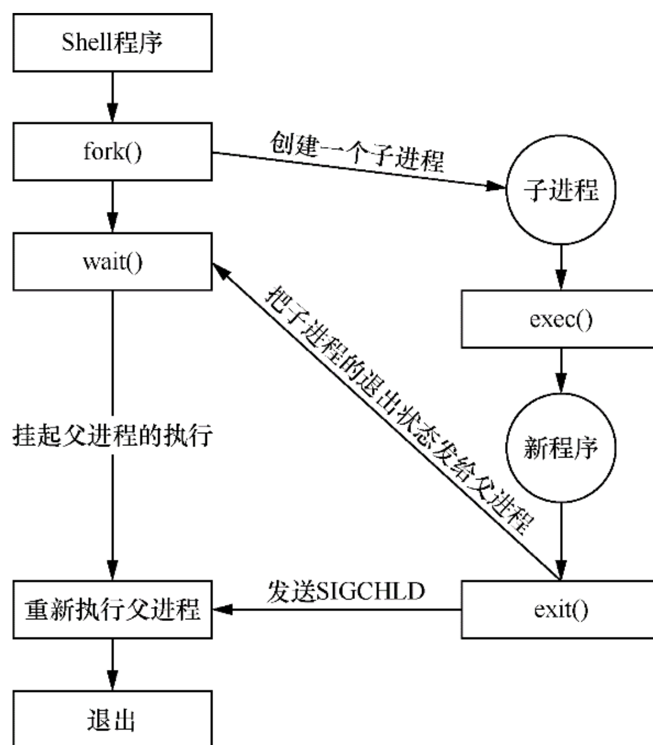

```
ppid     pid      pgid     sid
3512176 3514290 3514290 3512176 pts/19    3514364 RN     1000    0:34 ./proc1
3512176 3514291 3514290 3512176 pts/19    3514364 RN     1000    0:34 ./proc2
3512176 3514364 3514364 3512176 pts/19    3514364 R+     1000    0:15 ./proc3
3512176 3514365 3514364 3512176 pts/19    3514364 R+     1000    0:15 ./proc4
3512176 3514366 3514364 3512176 pts/19    3514364 R+     1000    0:15 ./proc5
```

遍历进程组和会话

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/sched/signal.h>

4.  static int __init pg_sid_init(void)
5.  {
6.      struct task_struct *task;
7.      for_each_process(task) {
8.          printk(KERN_INFO "PID=%d TGID=%d PGID=%d SID=%d\n",
9.                 task->pid, task->tgid, task->group_leader->tgid,
10.                task->sessionid);
11.     }
12.     return 0;
13. }

14. static void __exit pg_sid_exit(void)
15. {
16.     printk(KERN_INFO "Process group/session module unloaded\n");
17. }

18. module_init(pg_sid_init);
19. module_exit(pg_sid_exit);
20. MODULE_LICENSE("GPL");
```

3.2.3 进程的创建与终结

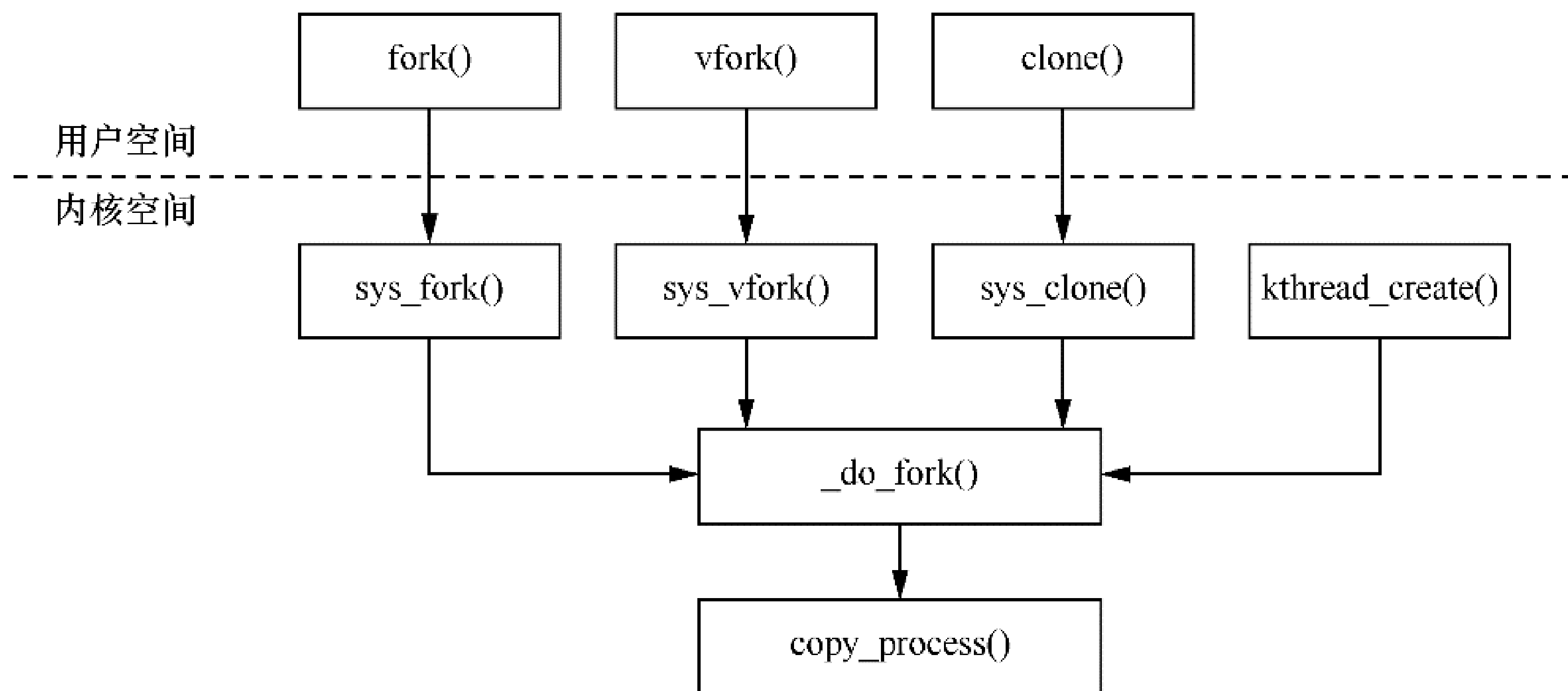
- 在Shell中执行命令的过程
  - Shell 调用 fork() 创建一个新的进程
  - 父进程调用 wait() 或 waitpid() 等待子进程退出，并获得其终止状态
  - 子进程调用 execve() 执行指定的程序，执行完毕后调用 exit() 退出



3.2.3 进程的创建与终结

- 进程创建原语：fork() 族函数、execve() 族函数
  - fork() 族函数通过写时复制技术复制当前进程的相关信息，以创建一个全新的子进程。这时子进程和父进程在各自的进程地址空间执行，拥有不同的pid，但共享相同的内容
  - execve() 族函数负责读取可执行文件，将其装入子进程的地址空间中并开始执行，这时父进程和子进程才开始分道扬镳
- 进程终止原语：wait() 族函数、exit() 族函数、kill() 族函数
  - exit() 族函数：终止当前进程并返回退出状态，释放资源并通知父进程。
  - wait() 族函数：父进程回收已终止子进程的资源，避免僵尸进程并获取退出状态。
  - kill() 族函数：向指定进程发送信号，常用于终止进程或控制进程行为。

进程创建原语：fork、vfork、clone



进程创建原语：fork、vfork、clone

```
// fork()为子进程建立了一个基于父进程的完整副本，为提高效率使用写时复制机制
SYSCALL_DEFINE0(fork) {  return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);  }

// vfork()的父进程会一直阻塞，直到子进程调用exit()或者execve()为止
SYSCALL_DEFINE0(vfork)  {
  return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,  0, NULL, NULL, 0);
}

// clone()通常用于创建用户线程，参数众多，可以有选择地继承父进程的资源
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
                                 int __user *, parent_tidptr, int __user *, child_tidptr, unsigned long, tls) {
  return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
}
```

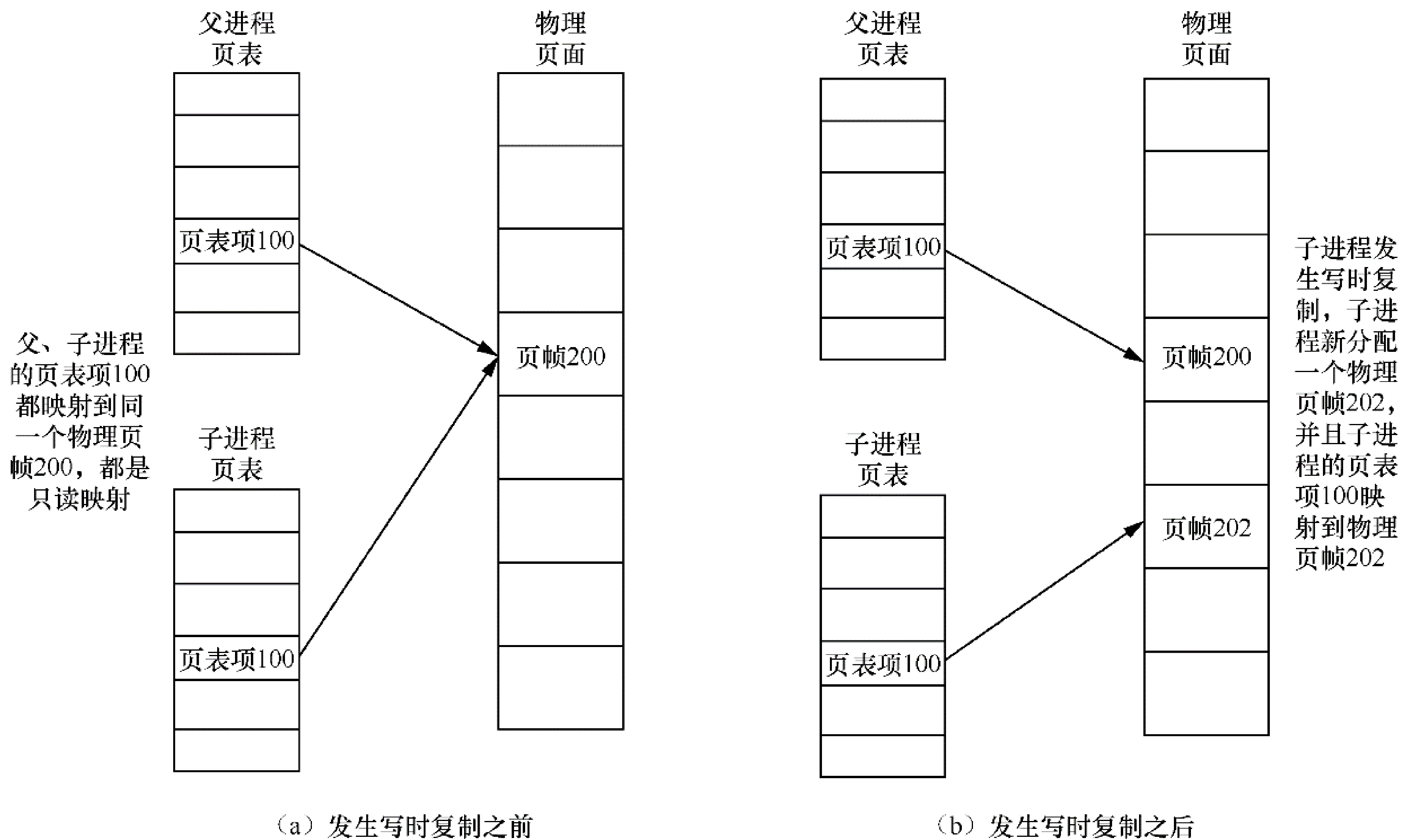
进程终结原语：exit、wait

- 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exit()系统调用。
  - 主动调用exit(0)系统调用
- 进程被动终止主要有如下3个途径。
  - 进程收到一个自己不能处理的信号。
  - 进程在内核态执行时产生了一个异常。
  - 进程收到SIGKILL等终止信号。
- 当一个进程终止时，Linux内核会释放它所占有的资源，并把这个消息告知父进程，而一个进程终止时可能有两种情况
  - 若它先于父进程终止，那么子进程会变成一个僵尸进程，直到父进程调用wait()才算最终消亡。
  - 若它也在父进程之后终止，那么init进程将成为子进程的新父进程。

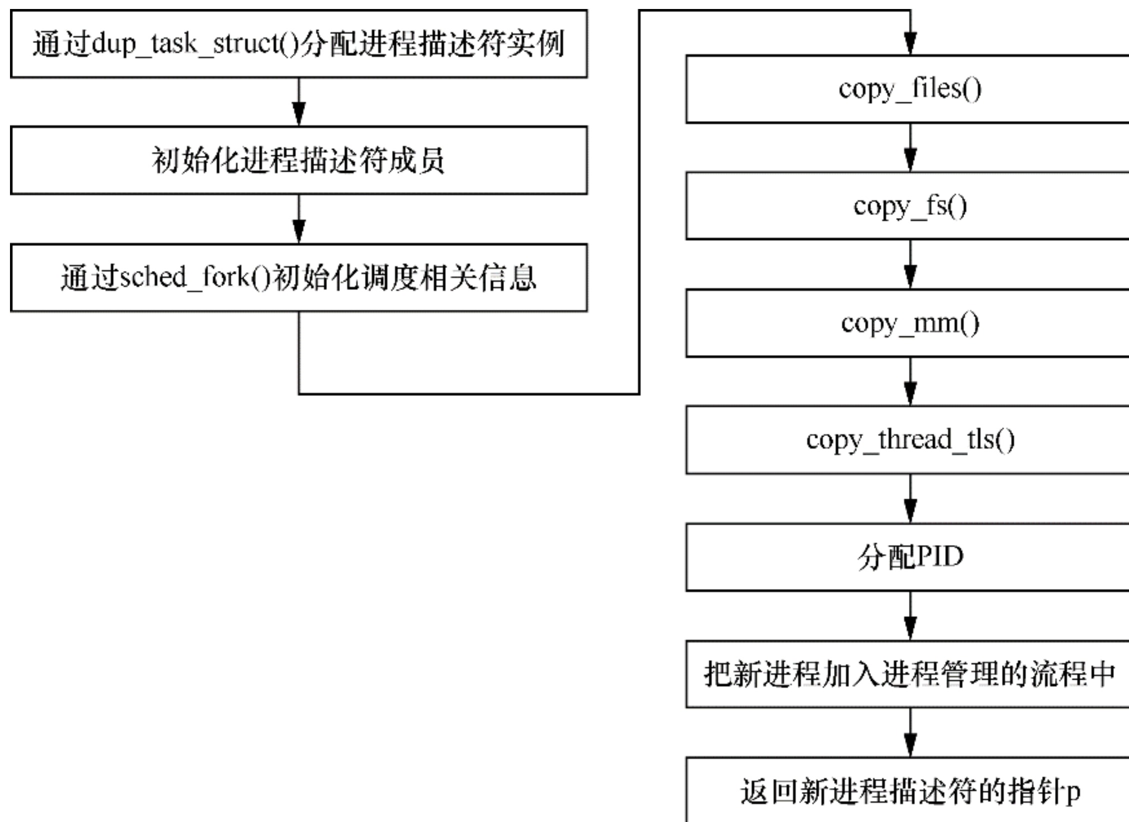
写时复制技术（Copy On Write，COW）

- 在传统的UNIX操作系统中，创建新进程时会复制父进程所拥有的所有资源，这样进程的创建变得很低效；现代操作系统都采用写时复制技术进行优化
- 父进程在创建子进程时无需复制进程地址空间的内容到子进程，只需要复制父进程地址空间的页表到子进程，这样父、子进程就共享了相同的物理内存
- 当父、子进程中有一方需要修改某个物理页面的内容时，触发写保护的缺页异常，然后才复制共享页面的内容，从而让父、子进程拥有各自的副本
- 父子进程的地址空间以只读方式共享，需要写入时才发生复制

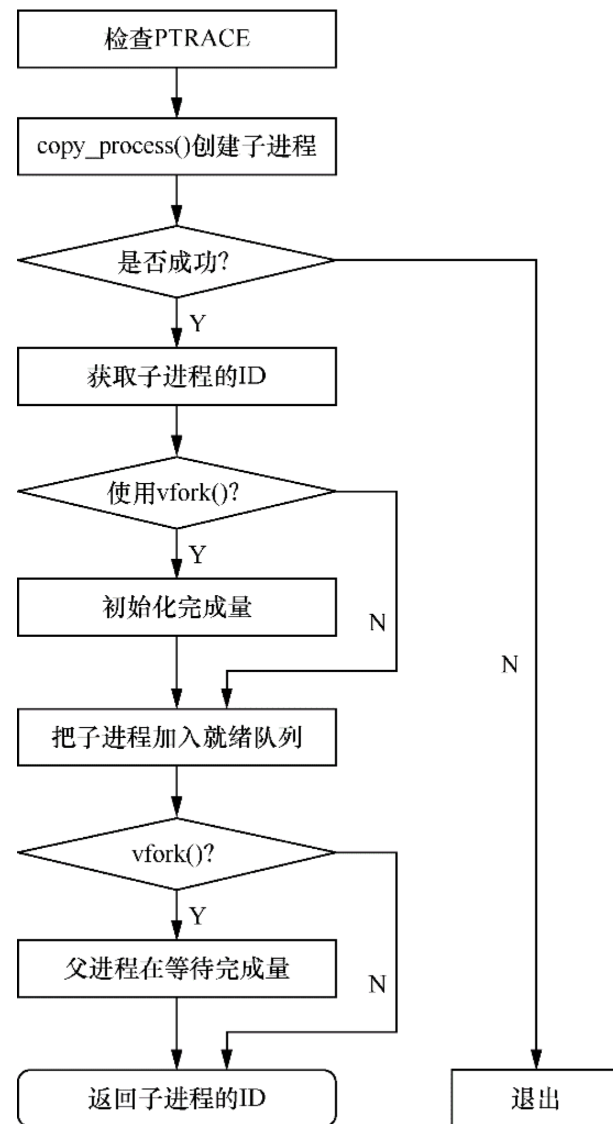
写时复制技术（Copy On Write，COW）



进程创建的内核源代码分析



copy_process()函数的流程



_do_fork()函数的流程

```c
long _do_fork(unsigned long clone_flags, unsigned long stack_start, unsigned long stack_size,
              int __user *parent_tidptr, int __user *child_tidptr, unsigned long tls)
{
    // 调用copy_process()函数创建一个新的子进程。如果创建成功，返回子进程的task_struct
    p = copy_process(clone_flags, stack_start, stack_size,
            child_tidptr, NULL, trace, tls, NUMA_NO_NODE);

    // 由子进程的task_struct数据结构来获取PID
    pid = get_task_pid(p, PIDTYPE_PID);
    nr = pid_vnr(pid);

    /*
      对于vfork()创建的子进程，首先要保证子进程先运行。在调用exec()或exit()之前，父、子进程是共享数据的。
      在子进程调用exec()或者exit()之后，才可以调度运行父进程，因此这里使用一个vfork_done完成量来达到扣留父进程的目的。
      init_completion()函数用于初始化这个完成量。
    */
    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
        get_task_struct(p);
    }

    // wake_up_new_task()函数唤醒新创建的进程，也就是把进程加入就绪队列里并接受调度、运行。
    wake_up_new_task(p);

    // 对于vfork()，wait_for_vfork_done()函数等待子进程调用exec()或者exit()。
    if (clone_flags & CLONE_VFORK) {
        if (!wait_for_vfork_done(p, &vfork))
            ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
    }

    // 在父进程返回用户空间时，其返回值为子进程的ID。子进程返回用户空间时，其返回值为0
    return nr;
}
```

```c
static struct task_struct *copy_process(unsigned long clone_flags, unsigned long stack_start,
                                        unsigned long stack_size, int __user *child_tidptr, struct pid *pid,
                                        int trace, unsigned long tls, int node) {
    // 为新进程分配一个task_struct数据结构
    p = dup_task_struct(current, node);

    // 复制父进程的证书
    copy_creds(p, clone_flags);

    // 初始化与调度相关的数据结构
    sched_fork(clone_flags, p);

    // 复制父进程打开的文件信息和文件系统信息
    copy_files(clone_flags, p);
    copy_fs(clone_flags, p);

    // 复制父进程的信号系统
    copy_sighand(clone_flags, p);
    copy_signal(clone_flags, p);

    // 复制父进程的进程地址空间的页表信息
    copy_mm(clone_flags, p);

    // 复制父进程的命名空间
    copy_namespaces(clone_flags, p);

    // 复制父进程中与I/Oi相关的内容
    copy_io(clone_flags, p);

    // 复制父进程人内核栈信息
    copy_thread_tls(clone_flags, stack_start, stack_size, p, tls);

    // 为新进程分配一个pid
    pid = alloc_pid(p->nsproxy->pid_ns_for_children);
    p->pid = pid_nr(pid);
    return p;
}
```

```c
static int copy_mm(unsigned long clone_flags, struct task_struct *tsk)
{
  struct mm_struct *mm, *oldmm;
  int retval;

  /*
    父进程的mm为空，说明父进程是一个没有用户态进程地址空间的内核线程
    这种情况下，无需为子进程做内存复制
  */
  oldmm = current->mm;
    if (!oldmm)
        return 0;

  /*
    若调用vfork()创建子进程，则CLONE_VM标志会被置位
    父子进程共享地址空间，只需让子进程的mm指针和父进程的mm指针值相同即可
  */
  if (clone_flags & CLONE_VM) {
    mmget(oldmm);
    mm = oldmm;
    goto good_mm;
  }

  /*
    若没有设置CLONE_VM标志位，则调用dup_mm将父进程的内存描述符的全部内容复制到子进程
    注意：这里仅仅复制内存描述符数据结构的内容，而不是复制整个内存（写时复制）
  */
  retval = -ENOMEM;
  mm = dup_mm(tsk);
  if (!mm)
    goto fail_nomem;

good_mm:
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;

fail_nomem:
    return retval;
}
```

3.2.4 Linux中的线程

- Linux中的线程本质是一个“轻量级”进程
  - 从内核的角度来说，它并没有线程这个概念。Linux 把所有的线程都当做进程来实现。
  - 内核并没有准备特别的调度算法或定义特别的数据结构来表征线程。相反，线程仅被视为一个与其他进程共享某些资源的进程。
  - 每个线程都拥有唯一隶属于自己的 task_struct，所以在内核中，它看起来就像是一个普通的进程，只是线程和其他一些进程共享某些资源。
- 内核线程
  - 内核经常需要在后台执行一些操作。这种任务可以通过内核线程完成。
  - 内核线程是独立运行在内核空间的标准进程，其与普通的进程间的区别在于内核线程没有独立的地空间（指向地址空间的 mm 指针被设置为NULL）。它们只在内核空间运行，从来不协到用户空间去。

3.3 进程调度

- 内存中保存了对每个进程的唯一描述，并通过若干结构与其他进程连接起来调度器面对的情形就是这样,其任务是在程序之间共享CPU时间,创造并行执行的错觉,该任务分为两个不同的部分,其中一个涉及调度策略,另外一个涉及上下文切换
- 内核必须提供一种方法,在各个进程之间尽可能公平地共享CPU时间,而同时又要考虑不同的任务优先级.
- 调度器的一个重要目标是有效地分配 CPU 时间片，同时提供很好的用户体验。调度器还需要面对-些互相冲突的目标，例如既要为关键实时任务最小化响应时间,又要最大限度地提高 CPU 的总体利用率
- 调度器的一般原理是,按所需分配的计算能力,向系统中每个进程提供最大的公正性,或者从另外一个角度上说，他试图确保没有进程被亏待.

3.3 进程调度

- linux把进程区分为实时进程和非实时进程,其中非实时进程进一步划分为交互式进程和批处理进程。

类型	描述	示例
交互式进程(interactive process)	此类进程经常与用户进行交互, 因此需要花费很多时间等待键盘和鼠标操作. 当接受了用户的输入后, 进程必须很快被唤醒, 否则用户会感觉系统反应迟钝	shell, 文本编辑程序和图形应用程序
批处理进程(batch process)	此类进程不必与用户交互, 因此经常在后台运行. 因为这样的进程不必很快相应, 因此常受到调度程序的怠慢	程序语言的编译程序, 数据库搜索引擎以及科学计算
实时进程(real-time process)	这些进程由很强的调度需要, 这样的进程绝不会被低优先级的进程阻塞. 并且他们的响应时间要尽可能的短	视频音频应用程序, 机器人控制程序以及从物理传感器上收集数据的程序

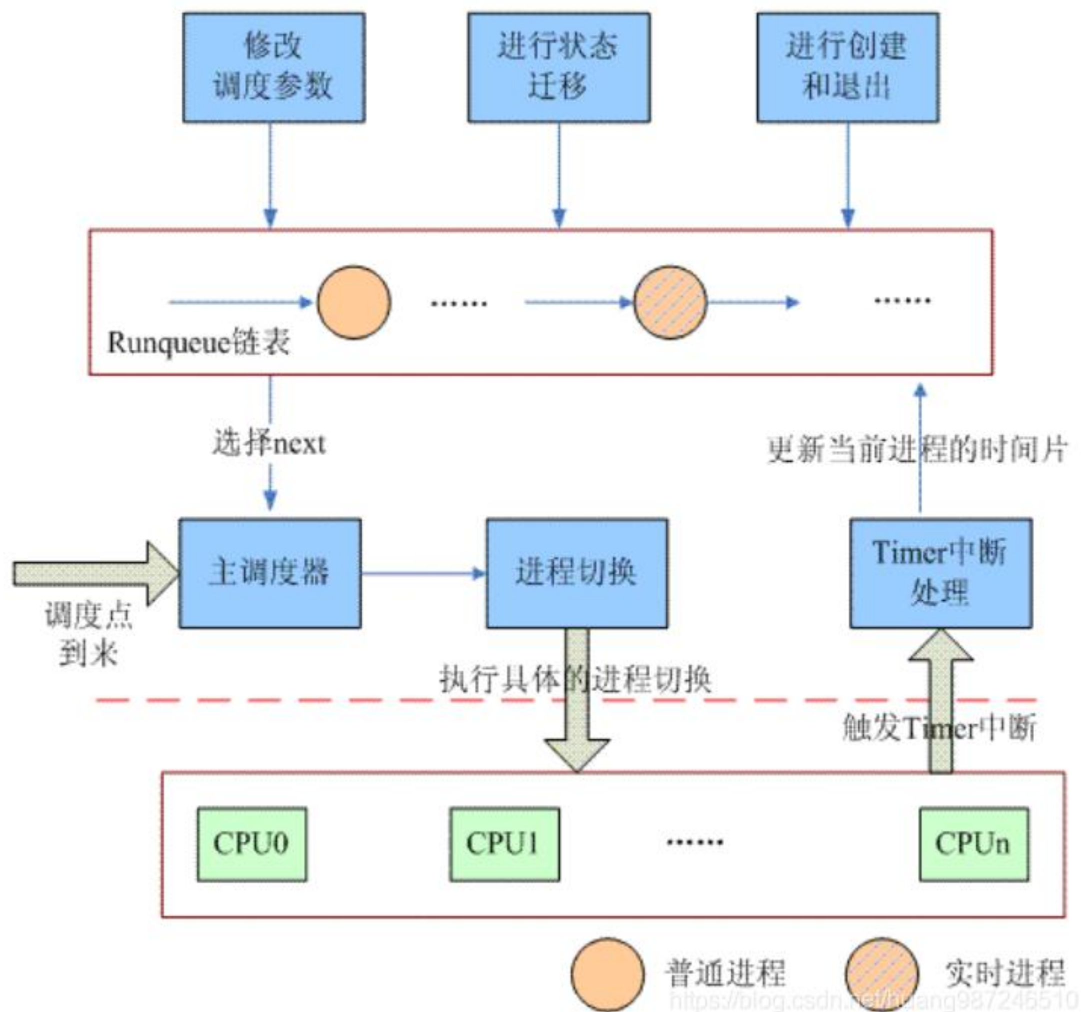
- 在linux中,调度算法可以明确的确认所有实时进程的身份,但是没办法区分交互式程序和批处理程序linux2.6的调度程序实现了基于进程过去行为的启发式算法,以确定进程应该被当做交互式进程还是批处理进程.当然与批处理进程相比,调度程序有偏爱交互式进程的倾向。

3.3 进程调度

- 根据进程的不同分类Linux采用不同的调度策略
  - 对于实时进程，采用FIFO或者Round Robin的调度策略
  - 对于普通进程，则需要区分交互式和批处理式的不同。传统Linux调度器提高交互式应用的优先级，使得它们能更快地被调度。而CFS和RSDL等新的调度器的核心思想是"完全公平"。这个设计理念不仅大大简化了调度器的代码复杂度，还对各种调度需求的提供了更完美的支持.
- 注意Linux通过将进程和线程调度视为一个，同时包含二者。进程可以看做是单个线程，但是进程可以包含共享一定资源(代码和/或数据)的多个线程。因此进程调度也包含了线程调度的功能
- 目前实时进程的调度策略比较简单,因为实时进程只要求尽可能快的被响应,基于优先级,每个进程根据它重要程度的不同被赋予不同的优先级，调度器在每次调度时,总选择优先级最高的进程开始执行.低优先级不可能抢占高优先级,因此FIFO或者Round Robin的调度策略即可满足实时进程调度的需求、
- 但是普通进程的调度策略就比较麻烦了,因为普通进程不能简单的只看优先级,必须公平的占有CPU.否则很容易出现进程饥饿,这种情况下用户会感觉操作系统很卡,响应总是很慢，因此在linux调度器的发展历程中经过了多次重大变动,inux总是希望寻找一个最接近于完美的调度策略来公平快速的调度进程.

O(n)调度器

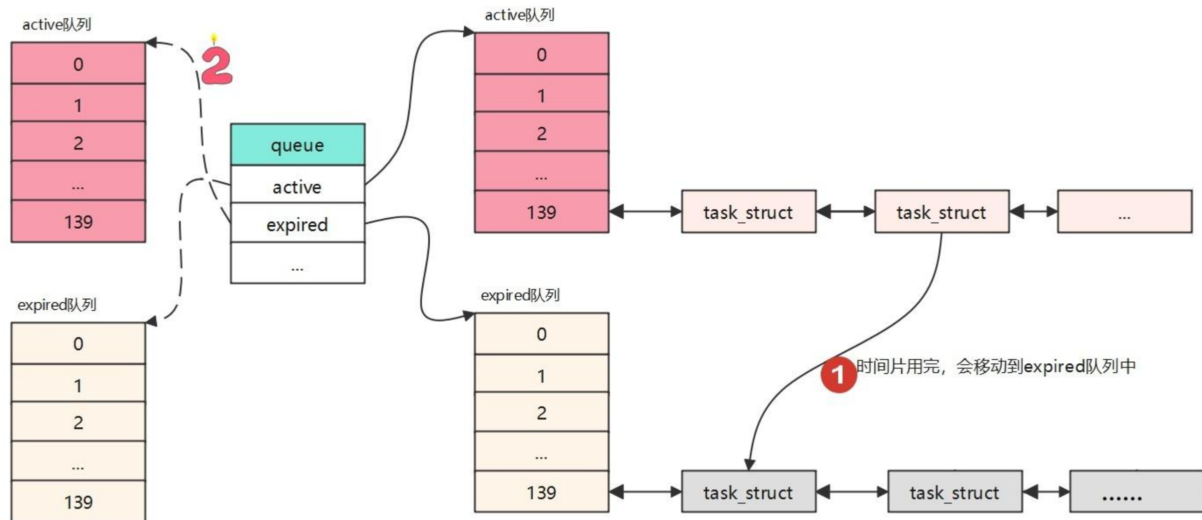
- 适用于Linux-2.6版本之前内核
- 采用一个Runqueue运行队列来管理所有可运行的进程
- 给每个进程赋予时间片，时间片的长短与进程优先级相关
- 一个进程的时间片用完了，调度器会选择下一个被调度的进程
- 在所有进程的时间片用完后，才重新计算任务的时间片
- 在主调度schedule()函数中遍历Runqueue运行队列，选择最优先级最高的进程运行

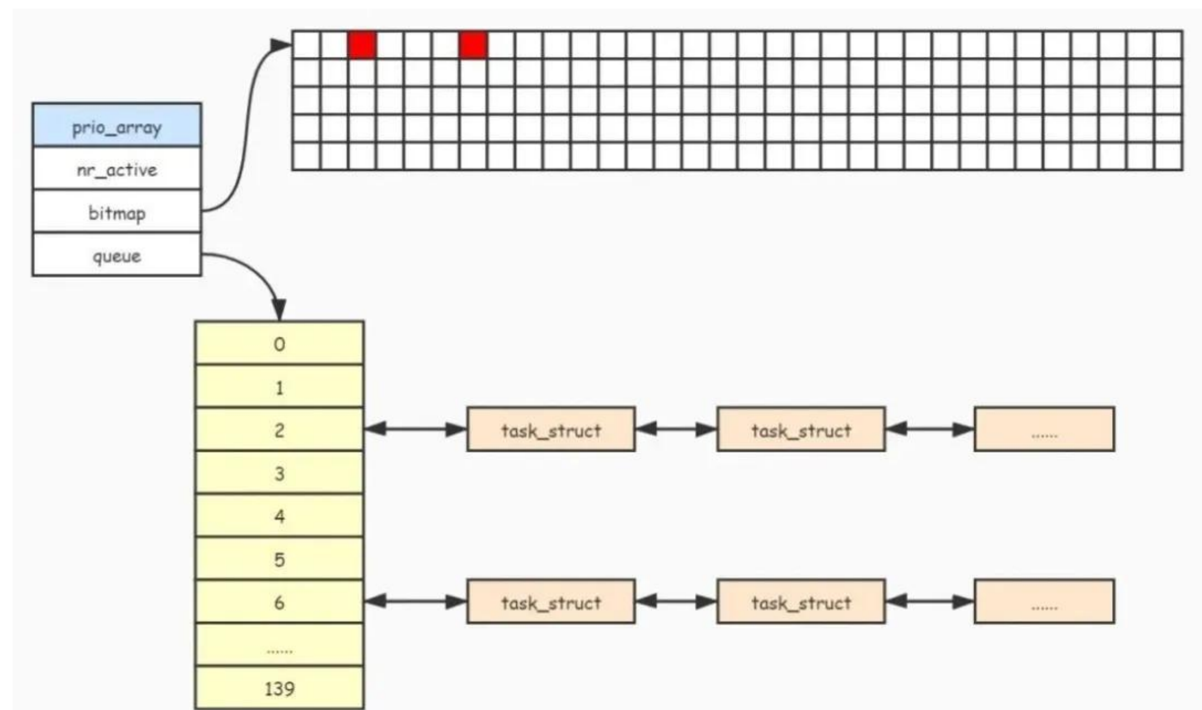


更多内容：https://blog.csdn.net/huang987246510/article/details/106130785/

O(1)调度器

- Linux-2.6版本内核引进
- 为每个CPU维护了两个队列：active队列存放时间片尚未用完的任务， expired队列存放时间片已经耗尽的任务
- 当一个任务的时间片用完后，就会被转到expired队列，且会重新计算它的优先级
- 当active队列任务全部转移到expired队列后，两个队列会一次进行交换
- 每个优先级维护了一个位图，能快速定位该优先级队列中的就绪进程





更多内容： https://zhuanlan.zhihu.com/p/489835657

3.3.1 Linux调度器的组成

- 2个调度器：调度器是统筹所有调度类的核心框架，负责协调不同策略的执行。
- 5个调度类：调度类是实现具体调度策略的模块，每个类对应一种或多种调度策略。
- 6种调度策略：调度策略是具体的规则或算法，用于决定 "何时选择哪个进程运行"。
- 3个调度实体
- 调度器整体框架

2个调度器

- 可以用两种方法来激活调度
  - 一种是直接的，比如进程打算睡眠或出于其他原因放弃CPU
  - 另一种是通过周期性的机制，以固定的频率运行，不时的检测是否有必要
- 因此当前linux的调度程序由两个调度器组成：主调度器，周期性调度器(两者又统称为通用调度器(generic scheduler)或核心调度器(core scheduler))
- 并且每个调度器包括两个内容：调度框架(其实质就是两个函数框架)及调度器类

5个调度类

调度器类	描述	对应调度策略
stop_sched_class	优先级最高的线程，会中断所有其他线程，且不会被其他任务打断 作用 1.发生在cpu_stop_cpu_callback 进行cpu之间任务migration 2.HOTPLUG_CPU的情况下关闭任务	无, 不需要调度普通进程
dl_sched_class	采用EDF最早截至时间优先算法调度实时进程	SCHED_DEADLINE
rt_sched_class	采用提供 Roound-Robin算法或者FIFO算法调度实时进程 具体调度策略由进程的task_struct->policy指定	SCHED_FIFO, SCHED_RR
fair_sched_clas	采用CFS算法调度普通的非实时进程	SCHED_NORMAL, SCHED_BATCH
idle_sched_class	采用CFS算法调度idle进程, 每个cup的第一个pid=0线程：swapper，是一个静态线程。调度类属于：idel_sched_class，所以在ps里面是看不到的。一般运行在开机过程和cpu异常的时候做dump	SCHED_IDLE

其所属进程的优先级顺序为

```
1  stop_sched_class -> dl_sched_class -> rt_sched_class -> fair_sched_class -> idle_sched_class
```

6种调度策略

- linux内核目前实现了6中调度策略(即调度算法), 用于对不同类型的进程进行调度, 或者支持某些特殊的功能

字段	描述	所在调度器类
SCHED_NORMAL	（也叫SCHED_OTHER）用于普通进程，通过CFS调度器实现。SCHED_BATCH用于非交互的处理器消耗型进程。SCHED_IDLE是在系统负载很低时使用	CFS
SCHED_BATCH	SCHED_NORMAL普通进程策略的分化版本。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。注意：这类进程比上述两类实时进程优先级低，换言之，在有实时进程存在时，实时进程优先调度。但针对吞吐量优化, 除了不能抢占外与常规任务一样，允许任务运行更长时间，更好地使用高速缓存，适合于成批处理的工作	CFS
SCHED_IDLE	优先级最低，在系统空闲时才跑这类进程(如利用闲散计算机资源跑地外文明搜索，蛋白质结构分析等任务，是此调度策略的适用者)	CFS-IDLE
SCHED_FIFO	先入先出调度算法（实时调度策略），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	RT
SCHED_RR	轮流调度算法（实时调度策略），后者提供 Roound-Robin 语义，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。不同要求的实时任务可以根据需要用sched_setscheduler() API设置策略	RT
SCHED_DEADLINE	新支持的实时进程调度策略，针对突发型计算，且对延迟和完成时间高度敏感的任务适用。基于Earliest Deadline First (EDF) 调度算法	DL

6种调度策略

// 作为系统初始化和管理进程，需公平共享CPU资源，无实时性要求，因此使用默认的完全公平调度（CFS）。
systemd (pid: 1) （policy: CFS)

// 内核线程migration/0负责CPU间任务迁移，需实时响应硬件中断和负载均衡，使用SCHED_FIFO确保不被抢占。
[3703468.775835] migration/0 (pid: 18) （policy: FIFO)

// 文件索引后台服务（如GNOME的Tracker），需在系统空闲时运行以减少资源争用，SCHED_IDLE策略赋予其最低优先级。
[3703468.777026] tracker-miner-f (pid: 3056222) （policy: IDLE)

// 用户自定义的实时任务，需时间片轮转（SCHED_RR）保证周期性执行，同时允许同优先级任务共享CPU。
[38.453278] set_rr (pid: 7853) （policy: RR)

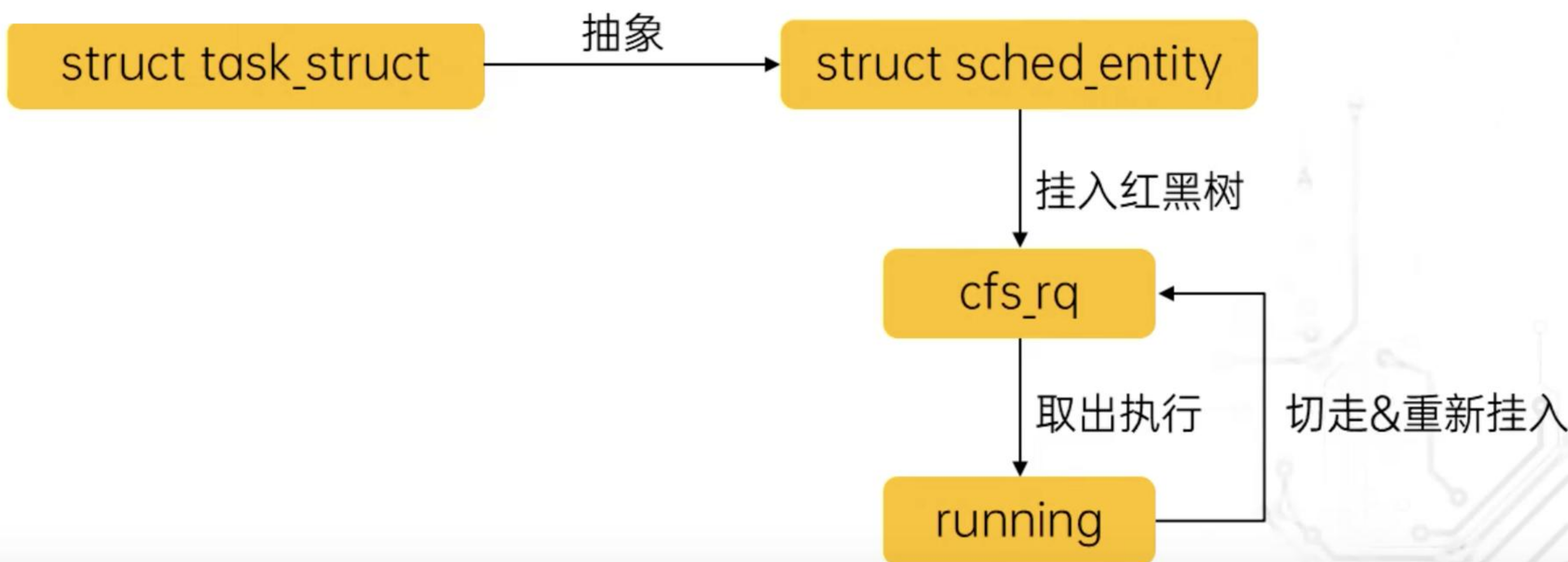
3个调度实体

- 调度器不限于调度进程，还可以调度更大的实体，比如实现组调度：可用的CPU时间首先在一半的进程组(比如，所有进程按照所有者分组)之间分配，接下来分配的时间再在组内进行二次分配。
- 这种一般性要求调度器不直接操作进程，而是处理可调度实体，因此需要一个通用的数据结构描述这个调度实体,即sched_entity结构，其实际上就代表了一个调度对象，可以为一个进程，也可以为一个进程组。
- linux中针对当前可调度的实时和非实时进程，定义了类型为sched_entity的3个调度实体。

调度实体	名称	描述	对应调度器类
sched_dl_entity	DEADLINE调度实体	采用EDF算法调度的实时调度实体	dl_sched_class
sched_rt_entity	RT调度实体	采用Roound-Robin或者FIFO算法调度的实时调度实体	rt_sched_class
sched_entity	CFS调度实体	采用CFS算法调度的普通非实时进程的调度实体	fair_sched_class

3个调度实体

- 对于单个进程来说，task_struct先被抽象为对应的调度实体sched_entity。调度实体作为基本单位被挂入运行队列cfs_rq。
- 之后每次从cfs_rq中取出一个实体/进程运行，运行之后若进程未执行完成则重新挂入cfs_rq。



调度器整体框架

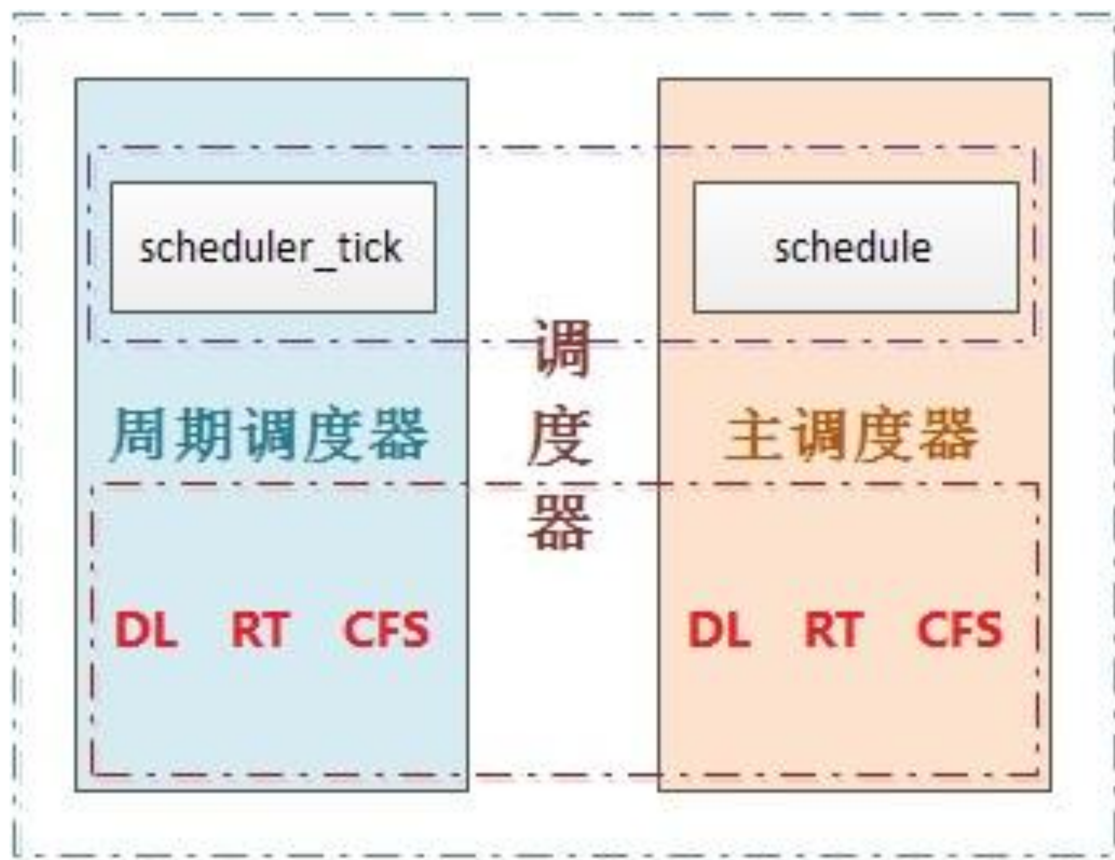
- 本质上, 通用调度器(核心调度器)是一个分配器,与其他两个组件交互.
  - 调度器用于判断接下来运行哪个进程， 内核支持不同的调度策略(完全公平调度, 实时调度, 在无事可做的时候调度空闲进程,即0号进程也叫swapper进程,idle进程), 调度类使得能够以模块化的方法实现这些侧露额, 即一个类的代码不需要与其他类的代码交互
  - 当调度器被调用时, 他会查询调度器类, 得知接下来运行哪个进程， 在选中将要运行的进程之后, 必须执行底层的任务切换， 这需要与CPU的紧密交互. 每个进程刚好属于某一调度类, 各个调度类负责管理所属的进程. 通用调度器自身不涉及进程管理, 其工作都委托给调度器类.
- 每个进程都属于某个调度器类(由字段task_struct->sched_class标识), 由调度器类采用进程对应的调度策略调度(由task_struct->policy)进行调度, task_struct也存储了其对应的调度实体标识

调度器整体框架

- linux实现了6种调度策略，依据其调度策略的不同实现了5个调度器类，一个调度器类可以用一种或者多种调度策略调度某一类进程，也可以用于特殊情况或者调度特殊功能的进程.

调度器类	调度策略	调度策略对应的调度算法	调度实体	调度实体对应的调度对象
stop_sched_class	无	无	无	特殊情况，发生在cpu_stop_cpu_callback 进行cpu之间任务迁移migration或者HOTPLUG_CPU的情况下关闭任务
dl_sched_class	SCHED_DEADLINE	Earliest-Deadline-First最早截至时间有限算法	sched_dl_entity	采用DEF最早截至时间有限算法调度实时进程
rt_sched_class	SCHED_RR SCHED_FIFO	Rooound-Robin时间片轮转算法 FIFO先进先出算法	sched_rt_entity	采用Rooound-Robin或者FIFO算法调度的实时调度实体
fair_sched_class	SCHED_NORMAL SCHED_BATCH	CFS完全公平懂调度算法	sched_entity	采用CFS算法普通非实时进程
idle_sched_class	SCHED_IDLE	无	无	特殊进程，用于cpu空闲时调度空闲进程idle

调度器整体框架



3.3.2 进程调度相关的数据结构

```
struct task_struct
{
    ........
    /* 表示是否在运行队列 */
    int on_rq;

    /* 进程优先级
     * prio: 动态优先级，范围为100~139，与静态优先级和补偿(bonus)有关
     * static_prio: 静态优先级，static_prio = 100 + nice + 20 (nice值为-20~19,所以static_prio值为100~139)
     * normal_prio: 没有受优先级继承影响的常规优先级，具体见normal_prio函数，跟属于什么类型的进程有关
     */
    int prio, static_prio, normal_prio;
    /* 实时进程优先级 */
    unsigned int rt_priority;

    /* 调度类，调度处理函数类 */
    const struct sched_class *sched_class;

    /* 调度实体(红黑树的一个结点) */
    struct sched_entity se;
    /* 调度实体(实时调度使用) */
    struct sched_rt_entity rt;
    struct sched_dl_entity dl;

#ifdef CONFIG_CGROUP_SCHED
    /* 指向其所在进程组 */
    struct task_group *sched_task_group;
#endif
    ........
}
```

优先级

```
int prio, static_prio, normal_prio;
unsigned int rt_priority;
```

动态优先级 静态优先级 实时优先级

其中task_struct采用了三个成员表示进程的优先级:prio和normal_prio表示动态优先级, static_prio表示进程的静态优先级.

> 为什么表示动态优先级需要两个值prio和normal_prio
> 调度器会考虑的优先级则保存在prio. 由于在某些情况下内核需要暂时提高进程的优先级, 因此需要用prio表示. 由于这些改变不是持久的, 因此静态优先级static_prio和普通优先级normal_prio不受影响.

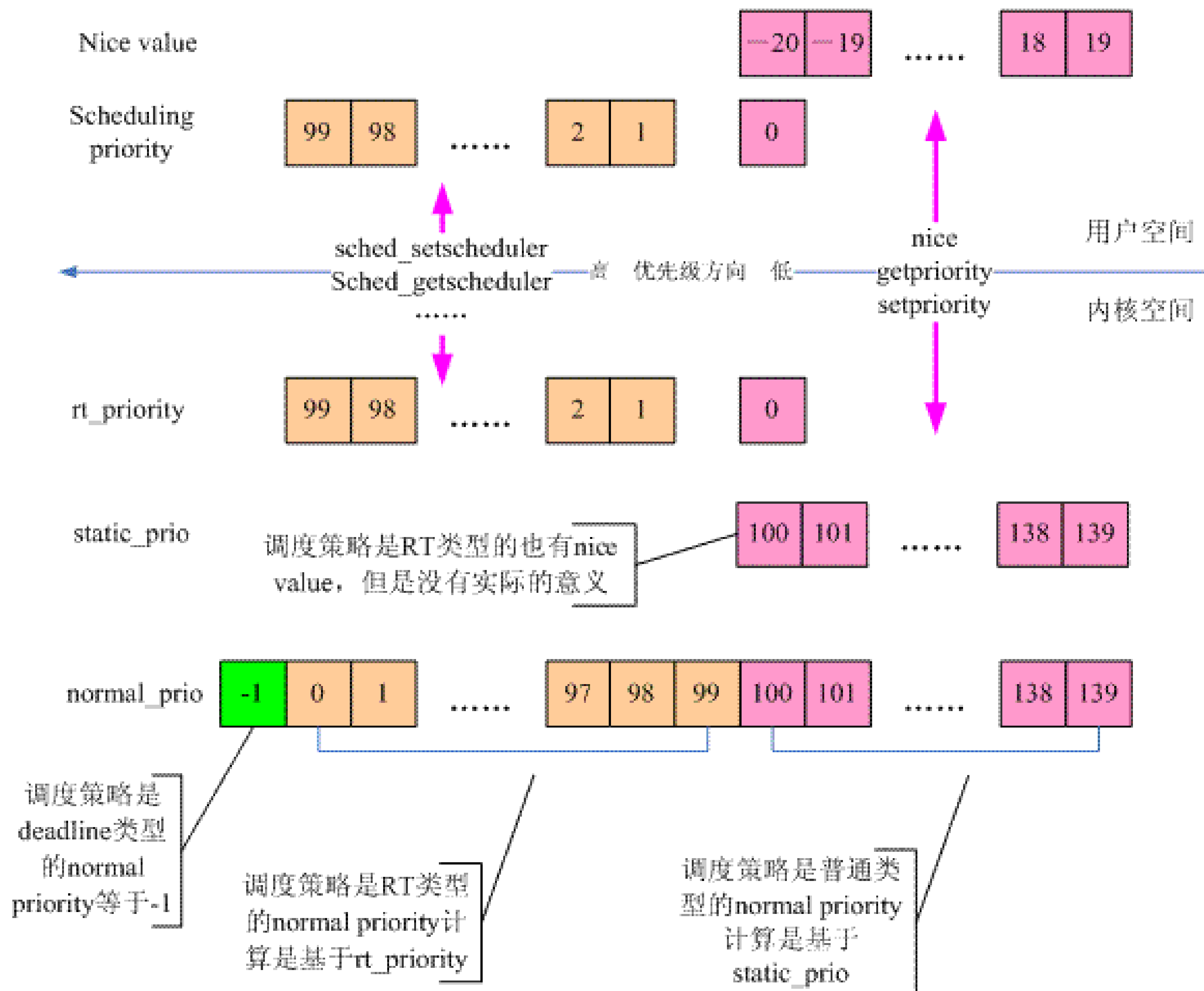
此外还用了一个字段rt_priority保存了实时进程的优先级

字段	描述
static_prio	用于保存静态优先级, 是进程启动时分配的优先级, , 可以通过nice和sched_setscheduler系统调用来进行修改, 否则在进程运行期间会一直保持恒定
prio	保存进程的动态优先级
normal_prio	表示基于进程的静态优先级static_prio和调度策略计算出的优先级. 因此即使普通进程和实时进程具有相同的静态优先级, 其普通优先级也是不同的, 进程分叉(fork)时, 子进程会继承父进程的普通优先级
rt_priority	用于保存实时优先级

实时进程的优先级用实时优先级rt_priority来表示

优先级

系统调用	描述
nice( )	设置进程的nice值
sched_setparam( )	设置进程的实时优先级
sched_getparam( )	获取进程的实时优先级
sched_get_priority_max( )	获取实时优先级的最大值
sched_get_priority_min( )	获取实时优先级的最小值



Nice value
-20
-19
....
18
19
Scheduling priority
99
98
2
1
0
sched_setscheduler
Sched_getscheduler
....
高 优先级方向 低
nice
getpriority
setpriority
用户空间
内核空间
rt_priority
99
98
2
1
0
static_prio
调度策略是RT类型的也有nice value，但是没有实际的意义
100
101
138
139
normal_prio
-1
0
1
97
98
99
100
101
138
139
调度策略是deadline类型的normal priority等于-1
调度策略是RT类型的normal priority计算是基于rt_priority
调度策略是普通类型的normal priority计算是基于static_prio

调度策略

```
unsigned int policy;
```

policy保存了进程的调度策略，目前主要有以下五种：

> 参见 http://lxr.free-electrons.com/source/include/uapi/linux/sched.h?v=4.6#L32

```
/*
* Scheduling policies
*/
#define SCHED_NORMAL            0
#define SCHED_FIFO              1
#define SCHED_RR                2
#define SCHED_BATCH             3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE              5
#define SCHED_DEADLINE          6
```

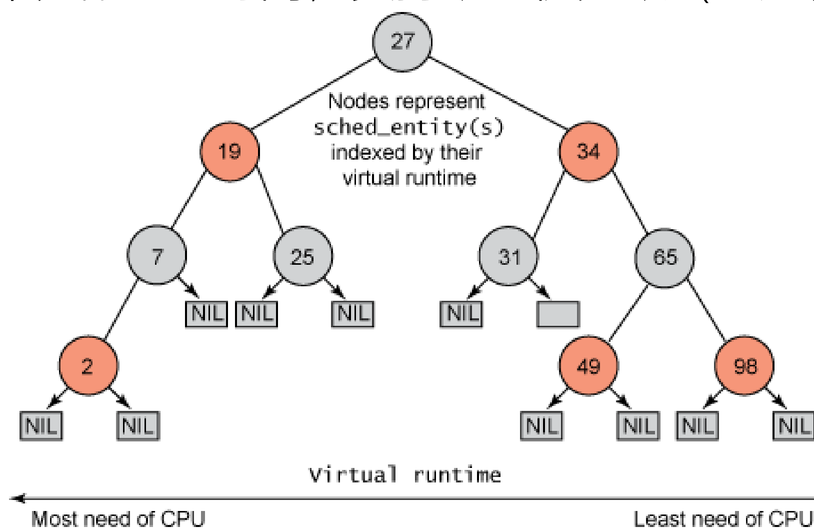
调度类

对于各个调度器类, 都必须提供struct sched_class的一个实例, 目前内核中有实现以下五种:

```
// http://lxr.free-electrons.com/source/kernel/sched/sched.h?v=4.6#L1254
extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;
```

CFS调度算法-CFS调度器的诞生背景与设计目标

- 前CFS调度器的不足
  - O(n)调度器：遍历所有进程计算优先级，时间复杂度O(n)，进程数量多时效率低下。
  - O(1)调度器：通过优先级数组实现快速选择，但依赖复杂的启发式规则，公平性不足。
- CFS的核心理念
  - 完全公平性：以虚拟运行时间（vruntime）为核心指标，而非优先级。
  - 设计目标：公平分配CPU时间，支持动态优先级（通过权重），保证低延迟和高吞吐量。



CFS调度算法-CFS的公平性模型与权重机制

- 从绝对公平到相对公平
  - 绝对公平问题：均分CPU时间忽略优先级差异（如后台任务与交互任务）。
  - 权重引入：通过进程的nice值映射权重，权重越高分配的CPU时间比例越大。
- 时间片分配公式
  - 时间片 = 调度周期 × (进程权重 / 总权重)
  - 示例：进程A（权重1）与B（权重2）在30ms调度周期中分得10ms和20ms。
- 权重与nice值的映射关系
  - prio_to_weight[40]数组，nice值每增减1，权重相差约10%（如nice=0 → 1024，nice=-20 → 88761）。

CFS调度算法-vruntime的核心作用

- vruntime的定义与计算
  - 公式：vruntime = 实际运行时间 × 1024 / 进程权重
  - 目的：标准化不同权重进程的CPU使用时间，实现公平比较。
- vruntime的调度决策
  - 红黑树中始终选择**vruntime最小的进程执行。**
  - 高权重进程优势：vruntime增长慢，长期获得更多CPU时间。
- 示例：
  - 进程A（权重1024）运行10ms → vruntime += 10。
  - 进程B（权重512）运行10ms → vruntime += 20。

[6105.947920][T8516] systemd: vruntime(1951464216647),weight(1048576)
[6105.948246][T8516] kthreadd: vruntime(1951398942314),weight(1048576)
[6105.948566][T8516] **pool_workqueue_: vruntime(8299189899),weight(1048576)**
优先运行

CFS调度算法-CFS的数据结构与调度流程

红黑树（RB-Tree）的核心作用

- 数据结构特性：自平衡二叉搜索树，插入、删除、查找最小值的时间复杂度为O(log n)。
- 多核扩展性：每个CPU维护独立红黑树，减少锁竞争。

调度周期与最小时间片

- 调度周期（sched_latency_ns）：默认6ms，确保所有就绪进程至少运行一次。
- 最小粒度（sched_min_granularity_ns）：默认0.75ms，避免频繁切换的开销。

调度流程详解:

- 步骤1：时钟中断触发，更新当前进程的vruntime。
- 步骤2：检查当前进程是否用完时间片或被更高优先级进程抢占。
- 步骤3：从红黑树中选择vruntime最小的进程，执行上下文切换。

CFS调度算法-进程生命周期管理

- 新进程的vruntime初始化
  - 防饿死机制：新进程的vruntime初始值 = 所在CPU的min_vruntime + 补偿值（如sched_vslice）。
    - sched_child_runs_first：确保子进程先于父进程运行（通过交换vruntime）。
    - START_DEBIT特性：增加新进程初始vruntime，防止fork炸弹攻击。
- 休眠进程的唤醒补偿
  - 补偿规则：唤醒时根据sysctl_sched_latency（默认6ms）调整vruntime，避免长期休眠进程饥饿。
- 进程迁移的vruntime同步
  - 跨CPU迁移公式：
    - 迁出时：vruntime -= 原队列min_vruntime
    - 迁入时：vruntime += 新队列min_vruntime
  - 目的：保持不同CPU队列中进程的vruntime公平性。

实践任务

- ProcessShow：添加一个内核模块，加载模块时打印内核中所有进程的状态。请修改这个模块，使得其能够打印更多的进程信息。
- ScheduleObserver：添加一个内核模块，加载模块时创建多个线程，线程中会让出CPU并记录被调用的次数，卸载模块时会打印每个线程被执行的次数。请修改线程0的调度方式，使得其被调用次数比其他线程多。

- 实验仓库链接：http://10.10.21.30/linux-kernel/practice_kern