# TODO

<http://gityuan.com/2016/09/17/android-lowmemorykiller/>

内存管理，lowmemorykill.c

<http://www.voidcn.com/article/p-pzlqoclp-bkg.html>

<http://gityuan.com/2016/09/17/android-lowmemorykiller/>

<https://cloud.tencent.com/developer/article/1029670>

# 一. 概述

Linux是类Unix系统，借鉴了Unix的设计并实现相关接口，但并非Unix。Linux是由Linus Torvalds于1991年创造的开源免费系统，采用GNU GPL协议保护，下面列举Linux的一些主要特点：

* Linux系统中万物皆为文件，这种抽象方便操作数据或设备，只需一套统一的系统接口open, read, write, close即可完成对文件的操作
* Linux是单内核，支持动态加载内核模块，可在运行时根据需求动态加载和卸载部分内核代码；
* Linux内核支持可抢占；
* Linux内核创建进程，采用独特的fork()系统调用，创建进程较高效；
* Linux内核并不区分进程和线程，对于内核而言，进程与线程无非是共享资源的区别，对CPU调度来说并没有显著差异。

**1.1 进程概念**

进程与线程的发展演化的目标，是为了更快的创建进程/线程，更小的上下文切换开销，更好的支持SMP以及HMP架构的CPU。 线程上下文(例如各个寄存器状态，pc指针)的切换比进程开销要小得多。

系统需要运转起来，代码都是静态的，进程才具有生命力，进程是程序的动态执行过程 ，进程正是操作系统的心脏所在。何为进程？进程是处于执行状态的代码以及相关资源的集合，不仅仅是代码段(text section)，还包括文件，信号，CPU状态，内存地址空间等。线程基本可以等同于进程般对待。

* 虚拟处理器：多个进程共享同一个处理器，但虚拟处理器给进程一种独占的感觉；
* 虚拟内存：多进程分享整个内存，但虚拟内存给进程以独占整个内存空间的感觉；

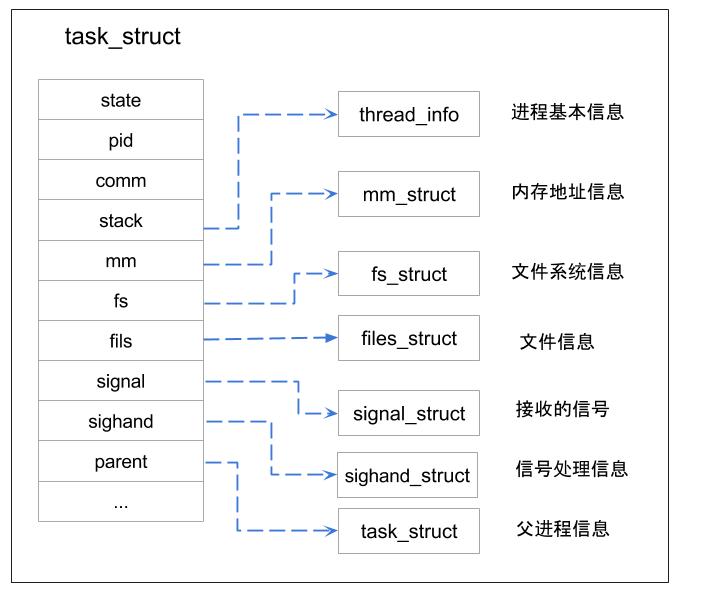
# 二. 进程

**2.1 task\_struct结构体**

进程主要由以下几部分组成：

* 代码段：编译后形成的一些指令
* 数据段：程序运行时需要的数据
  + 只读数据段：常量
  + 已初始化数据段：全局变量，静态变量
  + 未初始化数据段（bss)：未初始化的全局变量和静态变量
* 堆栈段：程序运行时动态分配的一些内存
* PCB：进程信息，状态标识等

Linux内核中进程用task\_struct结构体表示，称为进程描述符，该结构体相对比较复杂，有几百行代码，记载着该进程相关的所有信息，比如进程地址空间，进程状态，打开的文件等。对内核而言，进程或者线程都称为任务task。内核将所有进程放入一个双向循环链表结构的任务列表(task list)。



struct task\_struct {

volatile long state; //进程状态

struct mm\_struct \*mm, \*active\_mm; //内存地址空间

pid\_t pid;

pid\_t tgid;

struct task\_struct \_\_rcu \*real\_parent; //真正的父进程，fork时记录的

struct task\_struct \_\_rcu \*parent; // ptrace后，设置为trace当前进程的进程

struct list\_head children; //子进程

struct list\_head sibling; //父进程的子进程，即兄弟进程

struct task\_struct \*group\_leader; //线程组的领头线程

char comm[TASK\_COMM\_LEN]; //进程名，长度上限为16字符

struct fs\_struct \*fs; //文件系统信息

struct files\_struct \*files; // 打开的文件

struct signal\_struct \*signal;

struct sighand\_struct \*sighand;

struct sigpending pending;

void \*stack; // 指向内核栈的指针

...

}

进程运行在内核态时，需要相应的堆栈信息, 则linux kernel为每个进程都提供一个内核栈kernel stack.

**2.1.1 thread\_info**

Linux通过slab动态生成task\_struct，那么在栈顶或栈底创建新的结构体thread\_info即可，其中task指向其真正的task\_struct结构体。

struct thread\_info {

struct task\_struct \*task; //主要的进程描述符

struct exec\_domain \*exec\_domain;

\_\_u32 flags;

\_\_u32 status; // 线程同步flags

\_\_u32 cpu; //当前cpu

int preempt\_count;

mm\_segment\_t addr\_limit;

struct restart\_block restart\_block;

void \_\_user \*sysenter\_return;

unsigned int sig\_on\_uaccess\_error:1;

unsigned int uaccess\_err:1;

};

**2.2 进程状态**

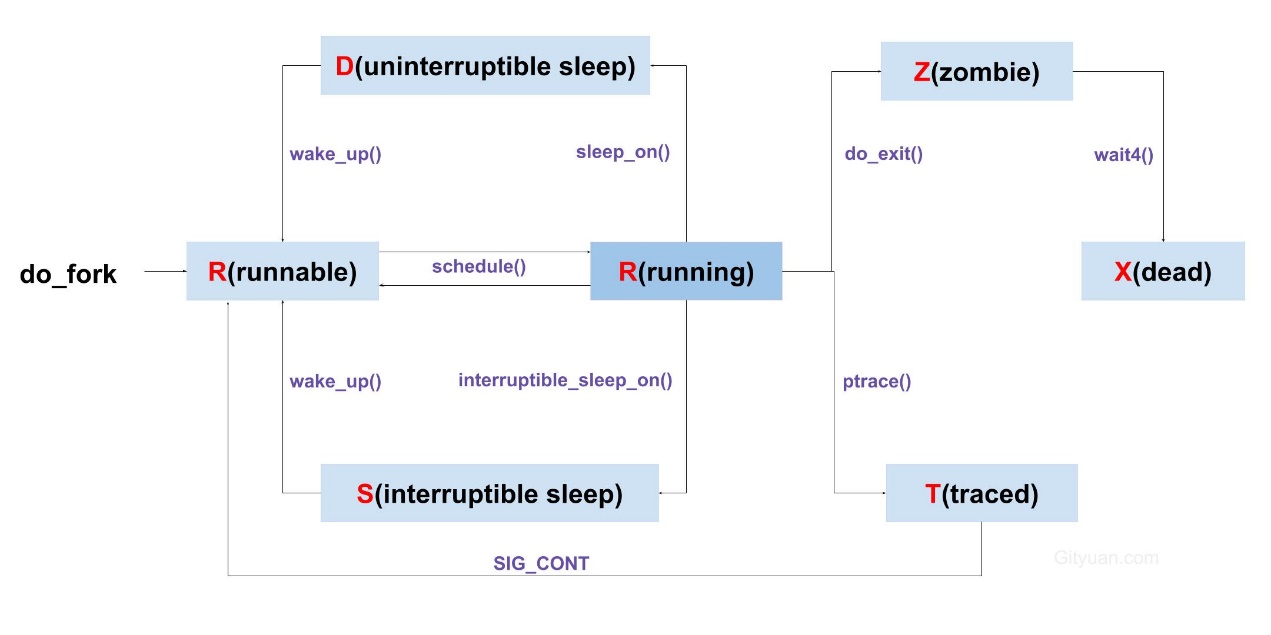
进程结构体task\_struct有一个成员state，代表的是进程的状态。 进程所有可能的状态定义在文件kernel/include/linux/sched.h， 不同的linux版本略有不同，下面列举最新Kernel的进程状态值：

| **序号** | **状态** | **缩写** | **含义** |
| --- | --- | --- | --- |
| 1 | TASK\_RUNNING | R | 正在运行或可运行 |
| 2 | TASK\_INTERRUPTIBLE | S | 可中断的休眠 |
| 3 | TASK\_UNINTERRUPTIBLE | D | 不可中断的休眠 |
| 4 | \_\_TASK\_STOPPED | T | 跟踪状态, 当进程接收到SIGSTOP等signal信息 |
| 5 | \_\_TASK\_TRACED | t | 停止状态，比如被debugger的ptrace() |
| 6 | EXIT\_ZOMBIE | Z | 僵尸状态，即父进程还没有执行waitpid() |
| 7 | EXIT\_DEAD | X | 死亡状态 |

说明：

* R状态: 分为正在执行和RQ队列等待执行两种状态，该状态是唯一可执行的状态；
* D状态：不影响任何信号，如果分析过一些系统冻屏/死机重启的案例，会发现很多时候是由于某个进程异常处于D状态而导致系统blocked。 即便如此，也有其存在的价值，比如当进程打开设备驱动文件时，在驱动程序执行完成之前是 不希望被打断的，可能会出现不可预知的状态。
* Z状态：出现这个状态往往是父进程没有执行waitpid()或wait4()系统调用， 在这种情况下，内核不会丢弃该死亡进程的信息，系统无法判断是父进程是否还需要该信息。

进程状态转换图：



**2.3 进程pid**

pid最大值默认为32768，一般来说pid数值越大的进程创建时间越晚，但进程再不断创建与结束，轮完一圈又会继续从小开始轮循，所以也就破坏了这个规则。可以通过修改/proc/sys/kernel/pid\_max来提高上限。

其他相关ID：

* Tgid: 线程组的ID,一个线程一定属于一个线程组(进程组).
* Pid: 进程的ID,更准确的说应该是线程的ID
* PPid: 当前进程的父进程

另外，每个进程的资源是有上限，可通过cat /proc//limits查看

**2.4 进程创建**

* Linux进程创建： 通过fork()系统调用创建进程
* Linux用户级线程创建：通过pthread库中的pthread\_create()创建线程
* Linux内核线程创建： 通过kthread\_create()创建内核线程

内核线程：没有独立的地址空间，即mm指向NULL。这样的线程只在内核运行，不会切换到用户空间。 所有内核线程都是由kthreadd作为内核线程的祖师爷，衍生而来的。

# Fork

*基于Kernel 4.4源码*

kernel/include/linux/sched.h

kernel/include/linux/kthread.h

kernel/arch/arm/include/asm/thread\_info.h

kernel/kernel/fork.c

kernel/kernel/exit.c

kernel/kernel/sched/core.c

**一. 概述**

Linux创建进程采用fork()和exec()

* fork: 采用复制当前进程的方式来创建子进程，此时子进程与父进程的区别仅在于pid, ppid以及资源统计量(比如挂起的信号)
* exec：读取可执行文件并载入地址空间执行；一般称之为exec函数族，有一系列exec开头的函数，比如execl, execve等

fork过程复制资源包括代码段，数据段，堆，栈。fork调用者所在进程便是父进程，新创建的进程便是子进程；在fork调用结束，从内核返回两次，一次继续执行父进程，一次进入执行子进程。

#### 1.1 进程创建

* Linux进程创建： 通过fork()系统调用创建进程
* Linux用户级线程创建：通过pthread库中的pthread\_create()创建线程
* Linux内核线程创建： 通过kthread\_create()

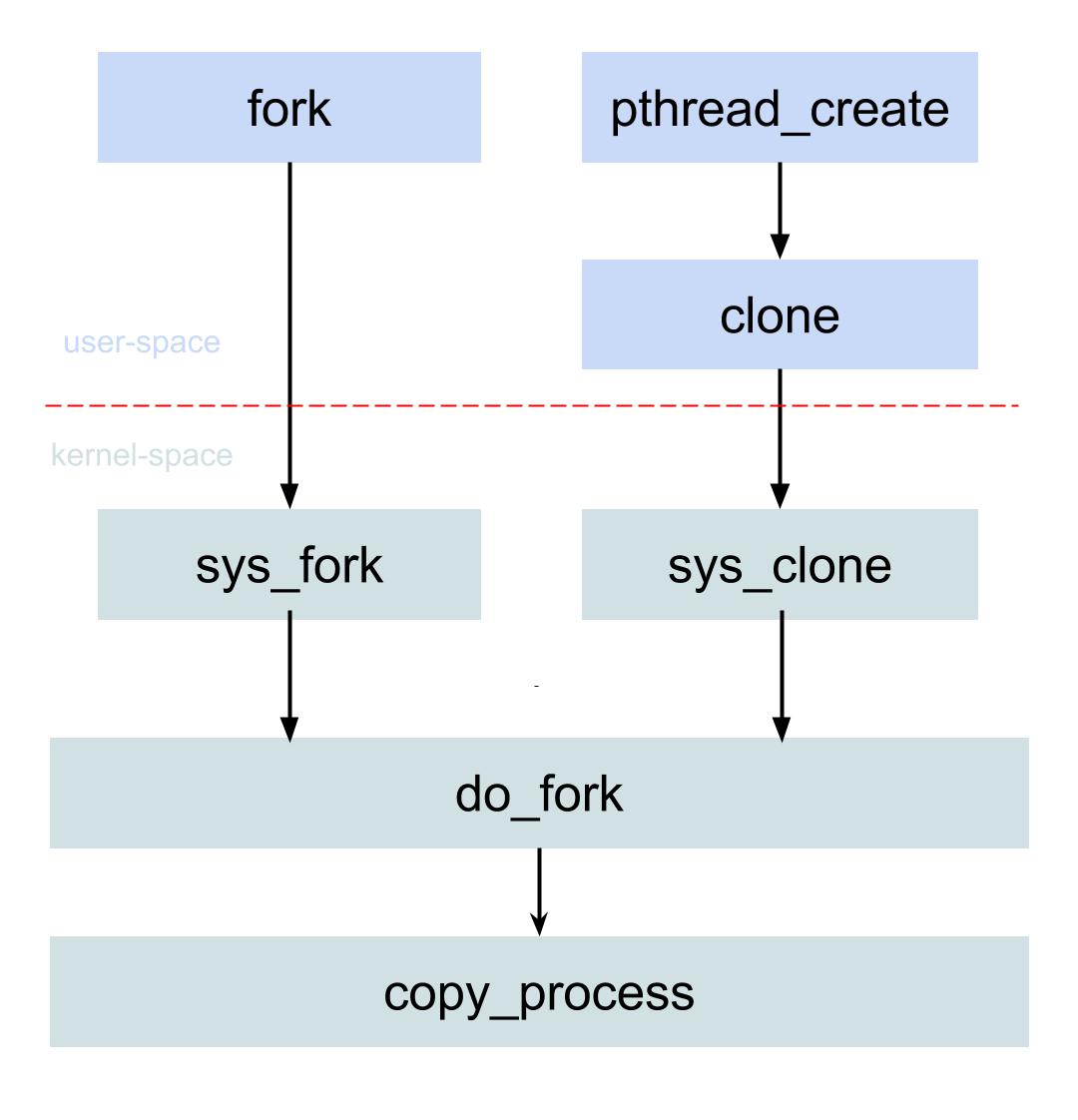
Linux线程，也并非”轻量级进程”，在Linux看来线程是一种进程间共享资源的方式，线程可看做是跟其他进程共享资源的进程。

fork, vfork, clone根据不同参数调用do\_fork

* pthread\_create: flags参数为 CLONE\_VM, CLONE\_FS, CLONE\_FILES, CLONE\_SIGHAND
* fork: flags参数为 SIGCHLD
* vfork: flags参数为 CLONE\_VFORK, CLONE\_VM, SIGCHLD

#### 1.2 fork流程图

进程/线程创建的方法fork(),pthread\_create(), 万物归一，最终在linux都是调用do\_fork方法。 当然还有vfork其实也是一样的， 通过系统调用到sys\_vfork，然后再调用do\_fork方法，该方法 现在很少使用，所以下图省略该方法。



fork执行流程:

1. 用户空间调用fork()方法;
2. 经过syscall陷入内核空间, 内核根据系统调用号找到相应的sys\_fork系统调用;
3. sys\_fork()过程会在调用do\_fork(), 该方法参数有一个flags很重要, 代表的是父子进程之间需要共享的资源; 对于进程创建flags=SIGCHLD, 即当子进程退出时向父进程发送SIGCHLD信号;
4. do\_fork(),会进行一些check过程,之后便是进入核心方法copy\_process.

#### 1.3 flags参数

进程与线程最大的区别在于资源是否共享，线程间共享的资源主要包括内存地址空间，文件系统，已打开文件，信号等信息， 如下图蓝色部分的flags便是线程创建过程所必需的参数。



fork采用Copy on Write机制，父子进程共用同一块内存，只有当父进程或者子进程执行写操作时会拷贝一份新内存。 另外，创建进程也是有可能失败，比如进程个数达到系统上限(32768)或者系统可用内存不足。

接下来进一步介绍fork过程。

## 二. fork源码分析

linux程序执行fork方法，通过中断(syscall)陷入内核，执行系统提供的相应系统调用来完成进程创建过程。

### 2.1 fork

[-> kernel/fork.c]

#### 2.1.1 fork

//fork系统调用

SYSCALL\_DEFINE0(fork)

{

return do\_fork(SIGCHLD, 0, 0, NULL, NULL);

}

#### 2.1.2 vfork

//vfork系统调用

SYSCALL\_DEFINE0(vfork)

{

return \_do\_fork(CLONE\_VFORK | CLONE\_VM | SIGCHLD, 0,

0, NULL, NULL, 0);

}

#### 2.1.3 clone

//clone系统调用

#ifdef CONFIG\_CLONE\_BACKWARDS

SYSCALL\_DEFINE5(clone, unsigned long, clone\_flags, unsigned long, newsp,

int \_\_user \*, parent\_tidptr,

unsigned long, tls,

int \_\_user \*, child\_tidptr)

#elif defined(CONFIG\_CLONE\_BACKWARDS2)

SYSCALL\_DEFINE5(clone, unsigned long, newsp, unsigned long, clone\_flags,

int \_\_user \*, parent\_tidptr,

int \_\_user \*, child\_tidptr,

unsigned long, tls)

#elif defined(CONFIG\_CLONE\_BACKWARDS3)

SYSCALL\_DEFINE6(clone, unsigned long, clone\_flags, unsigned long, newsp,

int, stack\_size,

int \_\_user \*, parent\_tidptr,

int \_\_user \*, child\_tidptr,

unsigned long, tls)

#else

SYSCALL\_DEFINE5(clone, unsigned long, clone\_flags, unsigned long, newsp,

int \_\_user \*, parent\_tidptr,

int \_\_user \*, child\_tidptr,

unsigned long, tls)

#endif

{

return \_do\_fork(clone\_flags, newsp, 0, parent\_tidptr, child\_tidptr, tls);

}

#endif

可见，fork,vfork,clone最终都会调用do\_fork方法，仅仅是flags不同。

### 2.2 do\_fork

[-> kernel/fork.c]

long do\_fork(unsigned long clone\_flags,

unsigned long stack\_start,

unsigned long stack\_size,

int \_\_user \*parent\_tidptr,

int \_\_user \*child\_tidptr)

{

return \_do\_fork(clone\_flags, stack\_start, stack\_size,

parent\_tidptr, child\_tidptr, 0);

}

参数说明：

* clone\_flags：clone方法传递过程的flags，标记子进程从父进程中需要继承的资源清单
* stack\_start: 子进程用户态的堆栈地址,fork()过程该值为0, clone()过程赋予有效值
* stack\_size：不必要的参数，默认设置为0
* parent\_tidptr：用户态下父进程的tid地址
* child\_tidptr：用户态下子进程的tid地址
* tls：

### 2.3 \_do\_fork

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)

{

struct task\_struct \*p;

...

//复制进程描述符【见小节2.4】

p = copy\_process(clone\_flags, stack\_start, stack\_size,

child\_tidptr, NULL, trace, tls);

if (!IS\_ERR(p)) {

struct completion vfork;

struct pid \*pid;

trace\_sched\_process\_fork(current, p);

//获取新创建的子进程的pid

pid = get\_task\_pid(p, PIDTYPE\_PID);

nr = pid\_vnr(pid);

if (clone\_flags & CLONE\_PARENT\_SETTID)

put\_user(nr, parent\_tidptr);

if (clone\_flags & CLONE\_VFORK) { //仅用于vfork过程,初始化

p->vfork\_done = &vfork;

init\_completion(&vfork);

get\_task\_struct(p);

}

wake\_up\_new\_task(p); //唤醒子进程，分配CPU时间片

if (unlikely(trace)) //告知ptracer，子进程已创建完成，并且已启动

ptrace\_event\_pid(trace, pid);

if (clone\_flags & CLONE\_VFORK) { //仅用于vfork过程，等待子进程

if (!wait\_for\_vfork\_done(p, &vfork))

ptrace\_event\_pid(PTRACE\_EVENT\_VFORK\_DONE, pid);

}

put\_pid(pid);

} else {

nr = PTR\_ERR(p);

}

return nr;

}

该方法过程说明：

1. 执行copy\_process，复制进程描述符，pid分配也是这个过程完成
2. 当为vfork，则执行相应的初始化过程
3. 执行wake\_up\_new\_task，唤醒子进程，分配CPU时间片
4. 当为vfork，则父进程等待子进程执行exec函数来替换地址空间

### 2.4 copy\_process

[-> kernel/fork.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 retval;

struct task\_struct \*p;

void \*cgrp\_ss\_priv[CGROUP\_CANFORK\_COUNT] = {};

...

retval = security\_task\_create(clone\_flags);

...

//【见小节2.4.1】

p = dup\_task\_struct(current);

...

rt\_mutex\_init\_task(p); //初始化mutex

//检查进程数是否超过上限

retval = -EAGAIN;

if (atomic\_read(&p->real\_cred->user->processes) >=

task\_rlimit(p, RLIMIT\_NPROC)) {

if (p->real\_cred->user != INIT\_USER &&

!capable(CAP\_SYS\_RESOURCE) && !capable(CAP\_SYS\_ADMIN))

goto bad\_fork\_free;

}

current->flags &= ~PF\_NPROC\_EXCEEDED;

retval = copy\_creds(p, clone\_flags);

...

//检查nr\_threads是否超过max\_threads

retval = -EAGAIN;

if (nr\_threads >= max\_threads)

goto bad\_fork\_cleanup\_count;

delayacct\_tsk\_init(p);

p->flags &= ~(PF\_SUPERPRIV | PF\_WQ\_WORKER);

p->flags |= PF\_FORKNOEXEC;

INIT\_LIST\_HEAD(&p->children);

INIT\_LIST\_HEAD(&p->sibling);

rcu\_copy\_process(p);

p->vfork\_done = NULL;

spin\_lock\_init(&p->alloc\_lock); //初始化自旋锁

init\_sigpending(&p->pending); //初始化挂起信号

p->utime = p->stime = p->gtime = 0;

p->utimescaled = p->stimescaled = 0;

prev\_cputime\_init(&p->prev\_cputime);

p->default\_timer\_slack\_ns = current->timer\_slack\_ns;

task\_io\_accounting\_init(&p->ioac);

acct\_clear\_integrals(p);

posix\_cpu\_timers\_init(p);

p->start\_time = ktime\_get\_ns(); //初始化进程启动时间

p->real\_start\_time = ktime\_get\_boot\_ns();

p->io\_context = NULL;

p->audit\_context = NULL;

cgroup\_fork(p);

p->pagefault\_disabled = 0;

//执行调度器相关设置，将该task分配给一某个CPU 【见小节2.4.2】

retval = sched\_fork(clone\_flags, p);

retval = perf\_event\_init\_task(p);

retval = audit\_alloc(p);

//拷贝进程的所有信息[见小节3.1]

shm\_init\_task(p);

retval = copy\_semundo(clone\_flags, p);

retval = copy\_files(clone\_flags, p);

retval = copy\_fs(clone\_flags, p);

retval = copy\_sighand(clone\_flags, p);

retval = copy\_signal(clone\_flags, p);

retval = copy\_mm(clone\_flags, p);

retval = copy\_namespaces(clone\_flags, p);

retval = copy\_io(clone\_flags, p);

retval = copy\_thread\_tls(clone\_flags, stack\_start, stack\_size, p, tls);

if (pid != &init\_struct\_pid) {

//分配pid[见小节2.4.3]

pid = alloc\_pid(p->nsproxy->pid\_ns\_for\_children);

...

}

...

p->pid = pid\_nr(pid); //设置pid

if (clone\_flags & CLONE\_THREAD) {

p->exit\_signal = -1;

p->group\_leader = current->group\_leader;

p->tgid = current->tgid;

} else {

if (clone\_flags & CLONE\_PARENT)

p->exit\_signal = current->group\_leader->exit\_signal;

else

p->exit\_signal = (clone\_flags & CSIGNAL);

p->group\_leader = p;

p->tgid = p->pid;

}

p->nr\_dirtied = 0;

p->nr\_dirtied\_pause = 128 >> (PAGE\_SHIFT - 10);

p->dirty\_paused\_when = 0;

p->pdeath\_signal = 0;

INIT\_LIST\_HEAD(&p->thread\_group);

p->task\_works = NULL;

threadgroup\_change\_begin(current);

...

//CLONE\_PARENT再利用旧的父进程

if (clone\_flags & (CLONE\_PARENT|CLONE\_THREAD)) {

p->real\_parent = current->real\_parent;

p->parent\_exec\_id = current->parent\_exec\_id;

} else {

p->real\_parent = current;

p->parent\_exec\_id = current->self\_exec\_id;

}

spin\_lock(&current->sighand->siglock);

copy\_seccomp(p);

recalc\_sigpending();

if (signal\_pending(current)) {

spin\_unlock(&current->sighand->siglock);

write\_unlock\_irq(&tasklist\_lock);

retval = -ERESTARTNOINTR;

goto bad\_fork\_cancel\_cgroup;

}

if (likely(p->pid)) {

ptrace\_init\_task(p, (clone\_flags & CLONE\_PTRACE) || trace);

init\_task\_pid(p, PIDTYPE\_PID, pid);

if (thread\_group\_leader(p)) {

init\_task\_pid(p, PIDTYPE\_PGID, task\_pgrp(current));

init\_task\_pid(p, PIDTYPE\_SID, task\_session(current));

if (is\_child\_reaper(pid)) {

ns\_of\_pid(pid)->child\_reaper = p;

p->signal->flags |= SIGNAL\_UNKILLABLE;

}

p->signal->leader\_pid = pid;

p->signal->tty = tty\_kref\_get(current->signal->tty);

list\_add\_tail(&p->sibling, &p->real\_parent->children);

list\_add\_tail\_rcu(&p->tasks, &init\_task.tasks);

attach\_pid(p, PIDTYPE\_PGID);

attach\_pid(p, PIDTYPE\_SID);

\_\_this\_cpu\_inc(process\_counts);

} else {

current->signal->nr\_threads++;

atomic\_inc(&current->signal->live);

atomic\_inc(&current->signal->sigcnt);

list\_add\_tail\_rcu(&p->thread\_group,

&p->group\_leader->thread\_group);

list\_add\_tail\_rcu(&p->thread\_node,

&p->signal->thread\_head);

}

attach\_pid(p, PIDTYPE\_PID);

nr\_threads++; //线程个数+1

}

total\_forks++; //进程forks次数加1

...

return p;

...

fork\_out:

return ERR\_PTR(retval);

}

主要功能：

1. 执行dup\_task\_struct()，拷贝当前进程task\_struct
2. 检查进程数是否超过系统所允许的上限(默认32678)
3. 执行sched\_fork()，设置调度器相关信息，设置task进程状态为TASK\_RUNNING，并分配CPU资源
4. 执行copy\_xxx()，拷贝进程的files, fs, mm, io, sighand, signal等信息
5. 执行copy\_thread\_tls(), 拷贝子进程的内核栈信息
6. 执行alloc\_pid()，为新进程分配新pid

#### 2.4.1 dup\_task\_struct

static struct task\_struct \*dup\_task\_struct(struct task\_struct \*orig) {

struct task\_struct \*tsk;

struct thread\_info \*ti;

int node = tsk\_fork\_get\_node(orig);

int err;

//分配task\_struct节点

tsk = alloc\_task\_struct\_node(node);

//分配thread\_info节点

ti = alloc\_thread\_info\_node(tsk, node);

err = arch\_dup\_task\_struct(tsk, orig);

//将thread\_info赋值给当前新创建的task

tsk->stack = ti;

setup\_thread\_stack(tsk, orig);

clear\_user\_return\_notifier(tsk);

clear\_tsk\_need\_resched(tsk);

set\_task\_stack\_end\_magic(tsk);

...

account\_kernel\_stack(ti, 1);

return tsk;

}

该方法主要功能是创建task\_struct和thread\_info结构体。执行完该方法，子进程唯一不同是tsk->stack。

#### 2.4.2 sched\_fork

[-> sched/core.c]

int sched\_fork(unsigned long clone\_flags, struct task\_struct \*p)

{

unsigned long flags;

int cpu = get\_cpu();

\_\_sched\_fork(clone\_flags, p);

//标记进程为Running状态，用于保证实际上并没有运行，

// 信号或许其他外部事件都无法唤醒该进程，同时把它插入运行队列

p->state = TASK\_RUNNING;

//确保不会将提升的优先级传递到子进程

p->prio = current->normal\_prio;

...

//为进程p分配相应cpu

set\_task\_cpu(p, cpu);

...

put\_cpu();

return 0;

}

#### 2.4.3 alloc\_pid

[-> kernel/kernel/pid.c]

struct pid \*alloc\_pid(struct pid\_namespace \*ns)

{

struct pid \*pid;

pid = kmem\_cache\_alloc(ns->pid\_cachep, GFP\_KERNEL);

...

tmp = ns;

pid->level = ns->level;

for (i = ns->level; i >= 0; i--) {

nr = alloc\_pidmap(tmp); //分配pid

...

pid->numbers[i].nr = nr; //nr保存到pid结构体

pid->numbers[i].ns = tmp;

tmp = tmp->parent;

}

...

return pid;

}

通过alloc\_pidmap()方法来完成pid的分配工作，具体分配算法见下一篇文章[Linux进程pid分配法](http://gityuan.com/2017/08/06/linux_process_pid/)

接下来的重头大戏是关于fs,mm等结构体的复制，见下面的过程。

## 三. 拷贝过程

### 3.1 copy\_semundo

[-> kernel/ipc/sem.c]

int copy\_semundo(unsigned long clone\_flags, struct task\_struct \*tsk)

{

struct sem\_undo\_list \*undo\_list;

int error;

if (clone\_flags & CLONE\_SYSVSEM) {

error = get\_undo\_list(&undo\_list);

if (error)

return error;

atomic\_inc(&undo\_list->refcnt);

tsk->sysvsem.undo\_list = undo\_list;

} else

tsk->sysvsem.undo\_list = NULL;

return 0;

}

当设置CLONE\_SYSVSEM，则父子进程间共享SEM\_UNDO状态

### 3.2 copy\_files

[-> fork.c]

static int copy\_files(unsigned long clone\_flags, struct task\_struct \*tsk)

{

struct files\_struct \*oldf, \*newf;

int error = 0;

//如果是后台进程可能没有files

oldf = current->files;

if (!oldf)

goto out;

//设置CLONE\_FILES，则只增加文件引用计数

if (clone\_flags & CLONE\_FILES) {

atomic\_inc(&oldf->count);

goto out;

}

//创建新的files\_struct，并拷贝内容【见小节3.2.1】

newf = dup\_fd(oldf, &error);

if (!newf)

goto out;

tsk->files = newf; //新创建的files\_struct赋值给新进程

error = 0;

out:

return error;

}

* 当父进程没有打开文件，则不需要执行文件拷贝
* 当设置CLONE\_FILES，则只增加文件引用计数，不创建新的files\_struct
* 以上都不满足，则创建新的files\_struct，并拷贝内容

#### 3.2.1 dup\_fd

[-> kernel/fs/file.c]

struct files\_struct \*dup\_fd(struct files\_struct \*oldf, int \*errorp)

{

//结构体成员【见小节3.2.2】

struct files\_struct \*newf;

struct file \*\*old\_fds, \*\*new\_fds;

int open\_files, i;

struct fdtable \*old\_fdt, \*new\_fdt;

//创建新files\_struct结构体

newf = kmem\_cache\_alloc(files\_cachep, GFP\_KERNEL);

...

atomic\_set(&newf->count, 1);

spin\_lock\_init(&newf->file\_lock);

newf->resize\_in\_progress = false;

init\_waitqueue\_head(&newf->resize\_wait);

newf->next\_fd = 0;

//初始化新的fdtable

new\_fdt = &newf->fdtab;

//NR\_OPEN\_DEFAULT等于BITS\_PER\_LONG，默认大小为32

new\_fdt->max\_fds = NR\_OPEN\_DEFAULT;

new\_fdt->close\_on\_exec = newf->close\_on\_exec\_init;

new\_fdt->open\_fds = newf->open\_fds\_init;

new\_fdt->full\_fds\_bits = newf->full\_fds\_bits\_init;

new\_fdt->fd = &newf->fd\_array[0];

spin\_lock(&oldf->file\_lock); //获取自旋锁

//获取oldf->fdt

old\_fdt = files\_fdtable(oldf);

//获取父进程(old\_fdt)所打开的文件个数 【见小节3.2.3】

open\_files = count\_open\_files(old\_fdt);

//当父进程打开的文件个数超过32个，则需要分配更大的fd数组/集合

while (unlikely(open\_files > new\_fdt->max\_fds)) {

spin\_unlock(&oldf->file\_lock); //释放自旋锁

if (new\_fdt != &newf->fdtab)

\_\_free\_fdtable(new\_fdt);

//分配更多的fdtable, 大小至少1KB且满足2的指数次方【见小节3.2.4】

new\_fdt = alloc\_fdtable(open\_files - 1);

...

//重新获取oldf锁，检查oldf最新打开的文件个数，再次检查是否小于max\_fds.

spin\_lock(&oldf->file\_lock);

old\_fdt = files\_fdtable(oldf);

open\_files = count\_open\_files(old\_fdt);

}

//拷贝父进程的fdtable信息【见小节3.2.5】

copy\_fd\_bitmaps(new\_fdt, old\_fdt, open\_files);

old\_fds = old\_fdt->fd;

new\_fds = new\_fdt->fd;

for (i = open\_files; i != 0; i--) {

struct file \*f = \*old\_fds++;

if (f) {

get\_file(f);

} else {

//fd已申明在fd数组，但是还没有文件open操作刚进行到一半，那么对于新进程不可用，则需要清除

\_\_clear\_open\_fd(open\_files - i, new\_fdt);

}

//先把内存写好，再把指针f赋值给new\_fds， rcu机制保证数据一致性

rcu\_assign\_pointer(\*new\_fds++, f);

}

spin\_unlock(&oldf->file\_lock);

//剩下的内存空间数据清零

memset(new\_fds, 0, (new\_fdt->max\_fds - open\_files) \* sizeof(struct file \*));

//将new\_fdt赋值给newf->fdt

rcu\_assign\_pointer(newf->fdt, new\_fdt);

return newf;

out\_release:

kmem\_cache\_free(files\_cachep, newf);

out:

return NULL;

}

该方法主要作用是创建和拷贝fdtable内容，然后赋值给新的file\_struct的成员指针fdt；

关于RCU的几个方法说明：

* rcu\_read\_lock：用于保护读者的RCU临界区，禁止抢占，不允许上下文切换；
* rcu\_read\_unlock：解除保护，恢复抢占
* rcu\_assign\_pointer：用于写者更新被RCU保护的指针
* rcu\_dereference：用于读者获取被RCU保护的指针
* synchronize\_rcu：等待之前所有的读取全部完成

要判断是不是被rcu\_read\_lock，可以观察是否发生了上下文切换(Context switch)；

#### 3.2.2 files\_struct结构体

[-> kernel/include/linux/fdtable.h]

struct files\_struct {

atomic\_t count;

bool resize\_in\_progress;

wait\_queue\_head\_t resize\_wait;

struct fdtable \_\_rcu \*fdt; //记录fdtable指针

struct fdtable fdtab; //记录fdtable

spinlock\_t file\_lock \_\_\_\_cacheline\_aligned\_in\_smp;

int next\_fd;

unsigned long close\_on\_exec\_init[1];

unsigned long open\_fds\_init[1];

unsigned long full\_fds\_bits\_init[1];

struct file \_\_rcu \* fd\_array[NR\_OPEN\_DEFAULT];

};

#define NR\_OPEN\_DEFAULT BITS\_PER\_LONG

struct fdtable {

unsigned int max\_fds;

struct file \_\_rcu \*\*fd; //当前fd数组

unsigned long \*close\_on\_exec;

unsigned long \*open\_fds; //打开的文件描述符

unsigned long \*full\_fds\_bits;

struct rcu\_head rcu;

};

#### 3.2.3 count\_open\_files

[-> kernel/fs/file.c]

***static int count\_open\_files(struct fdtable \*fdt) {***

***int size = fdt->max\_fds; //文件描述符的最大上限***

***int i;***

***//查询最后打开的fd, 其中BITS\_PER\_LONG=32***

***for (i = size / BITS\_PER\_LONG; i > 0; ) {***

***if (fdt->open\_fds[--i])***

***break;***

***}***

***i = (i + 1) \* BITS\_PER\_LONG;***

***return i;***

***}***

#### 3.2.4 alloc\_fdtable

[-> kernel/fs/file.c]

static struct fdtable \* alloc\_fdtable(unsigned int nr)

{

struct fdtable \*fdt;

void \*data;

//保证fd数组大小至少1KB，且fd个数是2的指数次方

nr /= (1024 / sizeof(struct file \*));

nr = roundup\_pow\_of\_two(nr + 1);

nr \*= (1024 / sizeof(struct file \*));

if (unlikely(nr > sysctl\_nr\_open))

nr = ((sysctl\_nr\_open - 1) | (BITS\_PER\_LONG - 1)) + 1;

//分配内存

fdt = kmalloc(sizeof(struct fdtable), GFP\_KERNEL);

fdt->max\_fds = nr;

data = alloc\_fdmem(nr \* sizeof(struct file \*));

fdt->fd = data;

data = alloc\_fdmem(max\_t(size\_t,

2 \* nr / BITS\_PER\_BYTE + BITBIT\_SIZE(nr), L1\_CACHE\_BYTES));

fdt->open\_fds = data;

data += nr / BITS\_PER\_BYTE;

fdt->close\_on\_exec = data;

data += nr / BITS\_PER\_BYTE;

fdt->full\_fds\_bits = data;

return fdt;

out\_arr:

kvfree(fdt->fd);

out\_fdt:

kfree(fdt);

out:

return NULL;

}

更新fdt的max\_fds，fd，open\_fds，close\_on\_exec，full\_fds\_bits数据。

#### 3.2.5 copy\_fd\_bitmaps

[-> file.c]

static void copy\_fd\_bitmaps(struct fdtable \*nfdt, struct fdtable \*ofdt, unsigned int count) {

unsigned int cpy, set;

cpy = count / BITS\_PER\_BYTE;

set = (nfdt->max\_fds - count) / BITS\_PER\_BYTE;

memcpy(nfdt->open\_fds, ofdt->open\_fds, cpy);

memset((char \*)nfdt->open\_fds + cpy, 0, set);

memcpy(nfdt->close\_on\_exec, ofdt->close\_on\_exec, cpy);

memset((char \*)nfdt->close\_on\_exec + cpy, 0, set);

cpy = BITBIT\_SIZE(count);

set = BITBIT\_SIZE(nfdt->max\_fds) - cpy;

memcpy(nfdt->full\_fds\_bits, ofdt->full\_fds\_bits, cpy);

memset((char \*)nfdt->full\_fds\_bits + cpy, 0, set);

}

该方法的功能：将ofdt的成员变量open\_fds和close\_on\_exec以及full\_fds\_bits数据拷贝到nfdt，没有数据的地方用0填充。

### 3.3 copy\_fs

[-> fork.c]

static int copy\_fs(unsigned long clone\_flags, struct task\_struct \*tsk)

{

//结构体【见小节3.3.1】

struct fs\_struct \*fs = current->fs;

if (clone\_flags & CLONE\_FS) {

spin\_lock(&fs->lock); //获取自旋锁

if (fs->in\_exec) {

spin\_unlock(&fs->lock);

return -EAGAIN;

}

fs->users++; //用户数加1

spin\_unlock(&fs->lock);

return 0;

}

//拷贝fs\_struct【见小节3.3.2】

tsk->fs = copy\_fs\_struct(fs);

if (!tsk->fs)

return -ENOMEM;

return 0;

}

该方法的功能：

* 当设置CLONE\_FS，且没有执行exec, 则设置用户数加1
* 当未设置CLONE\_FS，则拷贝fs\_struct结构体

#### 3.3.1 fs\_struct

[-> kernel/include/linux/fs\_struct.h]

struct fs\_struct {

int users;

spinlock\_t lock;

seqcount\_t seq;

int umask;

int in\_exec;

struct path root, pwd;

};

#### 3.3.2 copy\_fs\_struct

[-> kernel/fs/fs\_struct.c]

struct fs\_struct \*copy\_fs\_struct(struct fs\_struct \*old)

{

struct fs\_struct \*fs = kmem\_cache\_alloc(fs\_cachep, GFP\_KERNEL);

if (fs) {

fs->users = 1;

fs->in\_exec = 0;

spin\_lock\_init(&fs->lock);

seqcount\_init(&fs->seq);

fs->umask = old->umask;

spin\_lock(&old->lock);

fs->root = old->root;

path\_get(&fs->root);

fs->pwd = old->pwd;

path\_get(&fs->pwd);

spin\_unlock(&old->lock);

}

return fs;

}

### 3.4 copy\_sighand

[-> fork.c]

static int copy\_sighand(unsigned long clone\_flags, struct task\_struct \*tsk)

{

//【见小节3.4.1】

struct sighand\_struct \*sig;

if (clone\_flags & CLONE\_SIGHAND) {

atomic\_inc(&current->sighand->count);

return 0;

}

sig = kmem\_cache\_alloc(sighand\_cachep, GFP\_KERNEL);

rcu\_assign\_pointer(tsk->sighand, sig);

if (!sig)

return -ENOMEM;

atomic\_set(&sig->count, 1);

memcpy(sig->action, current->sighand->action, sizeof(sig->action));

return 0;

}

该方法的功能：

* 当设置CLONE\_SIGHAND, 则增加sighand->count计数
* 当未设置CLONE\_SIGHAND，则创建新的sighand\_struct结构体

#### 3.4.1 sighand\_struct结构体

[-> kernel/include/linux/sched.h]

struct sighand\_struct {

atomic\_t count; //计数

struct k\_sigaction action[\_NSIG];

spinlock\_t siglock;

wait\_queue\_head\_t signalfd\_wqh;

};

### 3.5 copy\_signal

[-> fork.c]

static int copy\_signal(unsigned long clone\_flags, struct task\_struct \*tsk)

{

//结构体【见小节3.5.1】

struct signal\_struct \*sig;

//当设置CLONE\_THREAD，则直接返回

if (clone\_flags & CLONE\_THREAD)

return 0;

//创建signal\_struct结构体

sig = kmem\_cache\_zalloc(signal\_cachep, GFP\_KERNEL);

tsk->signal = sig;

...

sig->nr\_threads = 1;

atomic\_set(&sig->live, 1);

atomic\_set(&sig->sigcnt, 1);

sig->thread\_head = (struct list\_head)LIST\_HEAD\_INIT(tsk->thread\_node);

tsk->thread\_node = (struct list\_head)LIST\_HEAD\_INIT(sig->thread\_head);

init\_waitqueue\_head(&sig->wait\_chldexit);

sig->curr\_target = tsk;

init\_sigpending(&sig->shared\_pending);

INIT\_LIST\_HEAD(&sig->posix\_timers);

seqlock\_init(&sig->stats\_lock);

prev\_cputime\_init(&sig->prev\_cputime);

hrtimer\_init(&sig->real\_timer, CLOCK\_MONOTONIC, HRTIMER\_MODE\_REL);

sig->real\_timer.function = it\_real\_fn;

task\_lock(current->group\_leader);

memcpy(sig->rlim, current->signal->rlim, sizeof sig->rlim);

task\_unlock(current->group\_leader);

posix\_cpu\_timers\_init\_group(sig);

tty\_audit\_fork(sig);

sched\_autogroup\_fork(sig);

//设置进程adj

sig->oom\_score\_adj = current->signal->oom\_score\_adj;

sig->oom\_score\_adj\_min = current->signal->oom\_score\_adj\_min;

sig->has\_child\_subreaper = current->signal->has\_child\_subreaper ||

current->signal->is\_child\_subreaper;

mutex\_init(&sig->cred\_guard\_mutex);

return 0;

}

#### 3.5.1 signal\_struct结构体

[-> kernel/include/linux/sched.h]

struct signal\_struct {

atomic\_t sigcnt;

atomic\_t live;

int nr\_threads;

struct list\_head thread\_head;

wait\_queue\_head\_t wait\_chldexit; //用于wait4()

struct task\_struct \*curr\_target; //当前线程组

struct sigpending shared\_pending; //共享信号处理

int group\_exit\_code; //线程组的退出码

//当通知完相应进程，则会唤醒group\_exit\_task进程

//当分发fatal信号，除了group\_exit\_task进程之外的都会被停止

int notify\_count;

struct task\_struct \*group\_exit\_task;

int group\_stop\_count;

unsigned int flags; //见SIGNAL\_\*系列

unsigned int is\_child\_subreaper:1;

unsigned int has\_child\_subreaper:1;

int posix\_timer\_id;

struct list\_head posix\_timers;

struct hrtimer real\_timer;

struct pid \*leader\_pid;

ktime\_t it\_real\_incr;

struct cpu\_itimer it[2];

struct thread\_group\_cputimer cputimer;

struct task\_cputime cputime\_expires;

struct list\_head cpu\_timers[3];

struct pid \*tty\_old\_pgrp;

int leader; //是否为对话组的领头线程

struct tty\_struct \*tty; //当没有tty，则为NULL

seqlock\_t stats\_lock;

cputime\_t utime, stime, cutime, cstime;

cputime\_t gtime;

cputime\_t cgtime;

struct prev\_cputime prev\_cputime;

unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw;

unsigned long min\_flt, maj\_flt, cmin\_flt, cmaj\_flt;

unsigned long inblock, oublock, cinblock, coublock;

unsigned long maxrss, cmaxrss;

struct task\_io\_accounting ioac;

unsigned long long sum\_sched\_runtime;

struct rlimit rlim[RLIM\_NLIMITS];

...

oom\_flags\_t oom\_flags;

short oom\_score\_adj; //OOM killer的adj值

short oom\_score\_adj\_min; //OOM killer的最小adj

struct mutex cred\_guard\_mutex;

};

signal\_struct结构体并没有自己的锁，而是利用sighand\_struct lock。

关于进程adj，可查看task->signal->oom\_score\_adj

### 3.6 copy\_mm

[-> fork.c]

static int copy\_mm(unsigned long clone\_flags, struct task\_struct \*tsk)

{

//结构体【见小节3.6.1】

struct mm\_struct \*mm, \*oldmm;

int retval;

tsk->min\_flt = tsk->maj\_flt = 0;

tsk->nvcsw = tsk->nivcsw = 0;

#ifdef CONFIG\_DETECT\_HUNG\_TASK

tsk->last\_switch\_count = tsk->nvcsw + tsk->nivcsw;

#endif

tsk->mm = NULL;

tsk->active\_mm = NULL;

//对于内核线程mm字段没空，则直接返回

oldmm = current->mm;

if (!oldmm)

return 0;

vmacache\_flush(tsk); //初始化新的vmacache实体

if (clone\_flags & CLONE\_VM) {

//增加引用计数

atomic\_inc(&oldmm->mm\_users);

mm = oldmm;

goto good\_mm;

}

retval = -ENOMEM;

//拷贝mm信息【见小节3.6.2】

mm = dup\_mm(tsk);

if (!mm)

goto fail\_nomem;

good\_mm:

tsk->mm = mm; //设置mm字段

tsk->active\_mm = mm;

return 0;

fail\_nomem:

return retval;

}

该方法说明：

* 对于内核线程mm字段没空，则直接返回
* 当设置CLONE\_VM，则增加mm\_users计数

#### 3.6.1 mm\_struct

[-> kernel/include/linux/mm\_types.h]

struct mm\_struct {

struct vm\_area\_struct \*mmap; //VMA列表

struct rb\_root mm\_rb;

u32 vmacache\_seqnum; //每个线程的vma缓存

#ifdef CONFIG\_MMU

unsigned long (\*get\_unmapped\_area) (struct file \*filp, unsigned long addr, unsigned long len, unsigned long pgoff, unsigned long flags);

#endif

unsigned long mmap\_base; //mmap区域

unsigned long mmap\_legacy\_base; //自下而上分配的mmap区域

unsigned long task\_size; //虚拟地址空间的大小

unsigned long highest\_vm\_end; //高端vma地址

pgd\_t \* pgd;

atomic\_t mm\_users; //使用该内存的进程个数

atomic\_t mm\_count; //结构体mm\_struct的引用个数

atomic\_long\_t nr\_ptes; //PTE页表

#if CONFIG\_PGTABLE\_LEVELS > 2

atomic\_long\_t nr\_pmds; //PMD页表

#endif

int map\_count; //VMA个数

spinlock\_t page\_table\_lock; //用于保活页表和一些计数

struct rw\_semaphore mmap\_sem;

struct list\_head mmlist;

unsigned long hiwater\_rss; //RSS的高水位使用情况

unsigned long hiwater\_vm; //高水位的虚拟内存使用情况

unsigned long total\_vm; //页面映射的总数

unsigned long locked\_vm; //PG\_mlocked的页面数

unsigned long pinned\_vm; //该计数永久增加

unsigned long shared\_vm; //共享页面数(files)

unsigned long exec\_vm; // VM\_EXEC & ~VM\_WRITE

unsigned long stack\_vm; // VM\_GROWSUP/DOWN

unsigned long def\_flags;

unsigned long start\_code, end\_code, start\_data, end\_data;

unsigned long start\_brk, brk, start\_stack;

unsigned long arg\_start, arg\_end, env\_start, env\_end;

unsigned long saved\_auxv[AT\_VECTOR\_SIZE]; //用于/proc/PID/auxv

struct mm\_rss\_stat rss\_stat;

struct linux\_binfmt \*binfmt;

cpumask\_var\_t cpu\_vm\_mask\_var;

mm\_context\_t context; //内存上下文

unsigned long flags;

struct core\_state \*core\_state; //支持coredump

#ifdef CONFIG\_AIO

spinlock\_t ioctx\_lock;

struct kioctx\_table \_\_rcu \*ioctx\_table;

#endif

#ifdef CONFIG\_MEMCG

struct task\_struct \_\_rcu \*owner;

#endif

/\* store ref to file /proc/<pid>/exe symlink points to \*/

struct file \_\_rcu \*exe\_file;

#ifdef CONFIG\_MMU\_NOTIFIER

struct mmu\_notifier\_mm \*mmu\_notifier\_mm;

#endif

#if defined(CONFIG\_TRANSPARENT\_HUGEPAGE) && !USE\_SPLIT\_PMD\_PTLOCKS

pgtable\_t pmd\_huge\_pte; /\* protected by page\_table\_lock \*/

#endif

#ifdef CONFIG\_CPUMASK\_OFFSTACK

struct cpumask cpumask\_allocation;

#endif

#ifdef CONFIG\_NUMA\_BALANCING

/\* \* numa\_next\_scan is the next time that the PTEs will be marked \* pte\_numa. NUMA hinting faults will gather statistics and migrate \* pages to new nodes if necessary. \*/

unsigned long numa\_next\_scan;

/\* Restart point for scanning and setting pte\_numa \*/

unsigned long numa\_scan\_offset;

int numa\_scan\_seq; //用于防止两个线程设置pte\_numa

#endif

#if defined(CONFIG\_NUMA\_BALANCING) || defined(CONFIG\_COMPACTION)

/\* \* An operation with batched TLB flushing is going on. Anything that \* can move process memory needs to flush the TLB when moving a \* PROT\_NONE or PROT\_NUMA mapped page. \*/

bool tlb\_flush\_pending;

#endif

struct uprobes\_state uprobes\_state;

#ifdef CONFIG\_X86\_INTEL\_MPX

void \_\_user \*bd\_addr; //绑定目录的地址

#endif

#ifdef CONFIG\_HUGETLB\_PAGE

atomic\_long\_t hugetlb\_usage;

#endif

};

#### 3.6.2 dup\_mm

[-> fork.c]

static struct mm\_struct \*dup\_mm(struct task\_struct \*tsk)

{

struct mm\_struct \*mm, \*oldmm = current->mm;

int err;

mm = allocate\_mm(); //分配内存

memcpy(mm, oldmm, sizeof(\*mm));

if (!mm\_init(mm, tsk)) //初始化mm

goto fail\_nomem;

err = dup\_mmap(mm, oldmm); //拷贝内存信息

if (err)

goto free\_pt;

mm->hiwater\_rss = get\_mm\_rss(mm);

mm->hiwater\_vm = mm->total\_vm;

if (mm->binfmt && !try\_module\_get(mm->binfmt->module))

goto free\_pt;

return mm;

free\_pt:

mm->binfmt = NULL;

mmput(mm);

fail\_nomem:

return NULL;

}

进程fork采用COW机制，实现的核心逻辑便在于内存拷贝过程会设置写保护。 具体实现在dup\_mmap()，这里暂不展开，后续再专门讲解。

### 3.7 copy\_namespaces

[-> kernel/kernel/nsproxy.c]

int copy\_namespaces(unsigned long flags, struct task\_struct \*tsk)

{

struct nsproxy \*old\_ns = tsk->nsproxy;

struct user\_namespace \*user\_ns = task\_cred\_xxx(tsk, user\_ns);

struct nsproxy \*new\_ns;

//一般情况都是进入该分支

if (likely(!(flags & (CLONE\_NEWNS | CLONE\_NEWUTS | CLONE\_NEWIPC |

CLONE\_NEWPID | CLONE\_NEWNET)))) {

get\_nsproxy(old\_ns);

return 0;

}

if (!ns\_capable(user\_ns, CAP\_SYS\_ADMIN))

return -EPERM;

if ((flags & (CLONE\_NEWIPC | CLONE\_SYSVSEM)) ==

(CLONE\_NEWIPC | CLONE\_SYSVSEM))

return -EINVAL;

//创建新的用户空间

new\_ns = create\_new\_namespaces(flags, tsk, user\_ns, tsk->fs);

...

tsk->nsproxy = new\_ns;

return 0;

}

#### 3.7.1 nsproxy结构体

[-> kernel/include/linux/nsproxy.h]

struct nsproxy {

atomic\_t count;

struct uts\_namespace \*uts\_ns;

struct ipc\_namespace \*ipc\_ns;

struct mnt\_namespace \*mnt\_ns;

struct pid\_namespace \*pid\_ns\_for\_children;

struct net \*net\_ns;

};

### 3.8 copy\_io

[-> fork.c]

***static int copy\_io(unsigned long clone\_flags, struct task\_struct \*tsk) {***

***#ifdef CONFIG\_BLOCK***

***struct io\_context \*ioc = current->io\_context;***

***struct io\_context \*new\_ioc;***

***if (!ioc)***

***return 0;***

***if (clone\_flags & CLONE\_IO) {***

***ioc\_task\_link(ioc); //nr\_tasks加1***

***tsk->io\_context = ioc;***

***} else if (ioprio\_valid(ioc->ioprio)) {***

***new\_ioc = get\_task\_io\_context(tsk, GFP\_KERNEL, NUMA\_NO\_NODE);***

***if (unlikely(!new\_ioc))***

***return -ENOMEM;***

***new\_ioc->ioprio = ioc->ioprio;***

***put\_io\_context(new\_ioc);***

***}***

***#endif***

***return 0;***

***}***

* 当设置CLONE\_IO，则父子进程间共享io context，nr\_tasks加1
* 否则，创建新io\_context结构体

#### 3.8.1 io\_context

[-> /kernel/include/linux/iocontext.h]

struct io\_context {

atomic\_long\_t refcount;

atomic\_t active\_ref;

atomic\_t nr\_tasks; //进程个数

spinlock\_t lock; //下面的成员都由该锁保护

unsigned short ioprio;

int nr\_batch\_requests; /\* Number of requests left in the batch \*/

unsigned long last\_waited; /\* Time last woken after wait for request \*/

struct radix\_tree\_root icq\_tree;

struct io\_cq \_\_rcu \*icq\_hint;

struct hlist\_head icq\_list;

struct work\_struct release\_work;

};

### 3.9 copy\_thread\_tls

[-> /kernel/arch/x86/kernel/process\_64.c]

copy\_thread\_tls(clone\_flags, stack\_start, stack\_size, p, tls);

int copy\_thread\_tls(unsigned long clone\_flags, unsigned long sp,

unsigned long arg, struct task\_struct \*p, unsigned long tls)

{

int err;

struct pt\_regs \*childregs;

struct task\_struct \*me = current;

//获取寄存器信息记录到thread\_struct结构体

p->thread.sp0 = (unsigned long)task\_stack\_page(p) + THREAD\_SIZE;

childregs = task\_pt\_regs(p);

p->thread.sp = (unsigned long) childregs;

set\_tsk\_thread\_flag(p, TIF\_FORK);

p->thread.io\_bitmap\_ptr = NULL;

savesegment(gs, p->thread.gsindex);

p->thread.gs = p->thread.gsindex ? 0 : me->thread.gs;

savesegment(fs, p->thread.fsindex);

p->thread.fs = p->thread.fsindex ? 0 : me->thread.fs;

savesegment(es, p->thread.es);

savesegment(ds, p->thread.ds);

memset(p->thread.ptrace\_bps, 0, sizeof(p->thread.ptrace\_bps));

if (unlikely(p->flags & PF\_KTHREAD)) {

//内核线程

memset(childregs, 0, sizeof(struct pt\_regs));

childregs->sp = (unsigned long)childregs;

childregs->ss = \_\_KERNEL\_DS;

childregs->bx = sp; /\* function \*/

childregs->bp = arg;

childregs->orig\_ax = -1;

childregs->cs = \_\_KERNEL\_CS | get\_kernel\_rpl();

childregs->flags = X86\_EFLAGS\_IF | X86\_EFLAGS\_FIXED;

return 0;

}

//当前寄存器数据复制给新创建的子进程

\*childregs = \*current\_pt\_regs();

//子进程eax设置为0，故fork在子进程返回值为0

childregs->ax = 0;

if (sp)

childregs->sp = sp;

err = -ENOMEM;

if (unlikely(test\_tsk\_thread\_flag(me, TIF\_IO\_BITMAP))) {

p->thread.io\_bitmap\_ptr = kmemdup(me->thread.io\_bitmap\_ptr,

IO\_BITMAP\_BYTES, GFP\_KERNEL);

if (!p->thread.io\_bitmap\_ptr) {

p->thread.io\_bitmap\_max = 0;

return -ENOMEM;

}

set\_tsk\_thread\_flag(p, TIF\_IO\_BITMAP);

}

//设置新的TLS

if (clone\_flags & CLONE\_SETTLS) {

err = do\_arch\_prctl(p, ARCH\_SET\_FS, tls);

if (err)

goto out;

}

err = 0;

out:

if (err && p->thread.io\_bitmap\_ptr) {

kfree(p->thread.io\_bitmap\_ptr);

p->thread.io\_bitmap\_max = 0;

}

return err;

}

设置子进程的寄存器等信息，从父进程拷贝thread\_struct的sp0，sp,io\_bitmap\_ptr等成员变量值。

## 四. 总结

流程：

do\_fork

\_do\_fork

copy\_process

dup\_task\_struct

sched\_fork

copy\_xxx

alloc\_pid

功能总结：

进程创建的核心实现在于copy\_process()方法过程，而copy\_process() 的主要实现在于copy\_xxx()方法，根据不同的flags来决策采用何种拷贝方式。

1. 执行dup\_task\_struct()，拷贝当前进程task\_struct
2. 检查进程数是否超过系统所允许的上限(默认32678)
3. 执行sched\_fork()，设置调度器相关信息，设置task进程状态为TASK\_RUNNING，并分配CPU资源
4. 执行copy\_xxx()，拷贝进程的相关资源信息
   * copy\_semundo: 当设置CLONE\_SYSVSEM，则父子进程间共享SEM\_UNDO状态
   * copy\_files: 当设置CLONE\_FILES，则只增加文件引用计数，不创建新的files\_struct
   * copy\_fs: 当设置CLONE\_FS，且没有执行exec, 则设置用户数加1
   * copy\_sighand: 当设置CLONE\_SIGHAND, 则增加sighand->count计数
   * copy\_signal: 拷贝进程信号
   * copy\_mm：当设置CLONE\_VM，则增加mm\_users计数
   * copy\_namespaces：一般情况，不需要创建新用户空间
   * copy\_io： 当设置CLONE\_IO，则父子进程间共享io context，增加nr\_tasks计数
   * copy\_thread\_tls：设置子进程的寄存器等信息，从父进程拷贝thread\_struct的sp0，sp,io\_bitmap\_ptr等成员变量值
5. 执行copy\_thread\_tls(), 拷贝子进程的内核栈信息
6. 执行alloc\_pid()，为新进程分配新pid

# Linux进程pid分配法

## 概述

Android系统创建进程，最终的实现还是调用linux fork方法，对于linux系统每个进程都有唯一的 进程ID(值大于0)，也有pid上限，默认为32768。 pid可重复利用，当进程被杀后会回收该pid，以供后续的进程pid分配。

上一篇文章[Linux进程管理](http://gityuan.com/2017/08/05/linux-process-fork/) 详细地介绍了进程fork过程，在copy\_process()过程，执行完父进行文件、内存等信息的拷贝，紧接着便是执行alloc\_pid()方法去分配pid.

## 二. 分配法

### 2.1 copy\_process

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) {

...

struct task\_struct \*p;

if (pid != &init\_struct\_pid) {

//分配pid[见小节2.2]

pid = alloc\_pid(p->nsproxy->pid\_ns\_for\_children);

}

p->pid = pid\_nr(pid); //设置pid[见小节2.4]

...

}

### 2.2 alloc\_pid

[-> kernel/kernel/pid.c]

struct pid \*alloc\_pid(struct pid\_namespace \*ns)

{

struct pid \*pid; //[见小节2.2.1]

enum pid\_type type;

int i, nr;

struct pid\_namespace \*tmp; //[见小节2.2.4]

struct upid \*upid;

int retval = -ENOMEM;

//分配pid结构体的内存

pid = kmem\_cache\_alloc(ns->pid\_cachep, GFP\_KERNEL);

...

tmp = ns;

pid->level = ns->level;

for (i = ns->level; i >= 0; i--) {

nr = alloc\_pidmap(tmp); //分配pid【见小节2.3】

...

pid->numbers[i].nr = nr; //nr保存到pid结构体

pid->numbers[i].ns = tmp;

tmp = tmp->parent;

}

...

get\_pid\_ns(ns);

atomic\_set(&pid->count, 1);

for (type = 0; type < PIDTYPE\_MAX; ++type)

INIT\_HLIST\_HEAD(&pid->tasks[type]); //初始化pid的hlist结构体

upid = pid->numbers + ns->level;

spin\_lock\_irq(&pidmap\_lock);

if (!(ns->nr\_hashed & PIDNS\_HASH\_ADDING))

goto out\_unlock;

for ( ; upid >= pid->numbers; --upid) {

//建立pid\_hash的关联关系

hlist\_add\_head\_rcu(&upid->pid\_chain,

&pid\_hash[pid\_hashfn(upid->nr, upid->ns)]);

upid->ns->nr\_hashed++;

}

spin\_unlock\_irq(&pidmap\_lock);

return pid;

...

}

#### 2.2.1 pid结构体

[-> kernel/include/linux/pid.h]

struct pid

{

atomic\_t count;

unsigned int level;

struct hlist\_head tasks[PIDTYPE\_MAX]; //见enum pid\_type

struct rcu\_head rcu;

struct upid numbers[1]; //见结构体upid

};

#### 2.2.2 upid结构体

[-> pid.h]

struct upid

{

int nr;

struct pid\_namespace \*ns;

struct hlist\_node pid\_chain;

};

#### 2.2.3 pid\_type

[-> pid.h]

enum pid\_type

{

PIDTYPE\_PID, //进程ID

PIDTYPE\_PGID, //进程组ID

PIDTYPE\_SID, //会话组ID

PIDTYPE\_MAX,

\_\_PIDTYPE\_TGID //仅用于\_\_task\_pid\_nr\_ns()

};

#### 2.2.4 pid\_namespace结构体

[-> kernel/include/linux/pid\_namespace.h]

struct pid\_namespace {

struct kref kref;

struct pidmap pidmap[PIDMAP\_ENTRIES];

struct rcu\_head rcu;

int last\_pid;

unsigned int nr\_hashed;

struct task\_struct \*child\_reaper;

struct kmem\_cache \*pid\_cachep;

unsigned int level;

struct pid\_namespace \*parent;

...

struct user\_namespace \*user\_ns;

struct work\_struct proc\_work;

kgid\_t pid\_gid;

int hide\_pid;

int reboot;

struct ns\_common ns;

};

PID命名空间，这是为系统提供虚拟化做支撑的功能。

### 2.3 alloc\_pidmap

[-> kernel/kernel/pid.c]

static int alloc\_pidmap(struct pid\_namespace \*pid\_ns)

{

//last\_pid为上次分配出去的pid

int i, offset, max\_scan, pid, last = pid\_ns->last\_pid;

struct pidmap \*map;

pid = last + 1;

if (pid >= pid\_max)

pid = RESERVED\_PIDS; //默认为300

offset = pid & BITS\_PER\_PAGE\_MASK; //最高位值置0，其余位不变

map = &pid\_ns->pidmap[pid/BITS\_PER\_PAGE]; //找到目标pidmap

//当offset =0，则扫描一次;

//当offset!=0，则扫描两次

max\_scan = DIV\_ROUND\_UP(pid\_max, BITS\_PER\_PAGE) - !offset;

for (i = 0; i <= max\_scan; ++i) {

if (unlikely(!map->page)) {

void \*page = kzalloc(PAGE\_SIZE, GFP\_KERNEL);

spin\_lock\_irq(&pidmap\_lock);

if (!map->page) {

map->page = page;

page = NULL;

}

spin\_unlock\_irq(&pidmap\_lock);

kfree(page);

if (unlikely(!map->page))

break;

}

//当pidmap还有可用pid时

if (likely(atomic\_read(&map->nr\_free))) {

do {

//当offset位空闲时返回该pid

if (!test\_and\_set\_bit(offset, map->page)) {

atomic\_dec(&map->nr\_free); //可用pid减一

set\_last\_pid(pid\_ns, last, pid); //设置last\_pid

return pid;

}

//否则，查询下一个非0的offset值

offset = find\_next\_offset(map, offset);

根据offset转换成相应的pid

pid = mk\_pid(pid\_ns, map, offset);

} while (offset < BITS\_PER\_PAGE && pid < pid\_max);

}

//当上述pid分配失败，则再次查找offset

if (map < &pid\_ns->pidmap[(pid\_max-1)/BITS\_PER\_PAGE]) {

++map;

offset = 0;

} else {

map = &pid\_ns->pidmap[0];

offset = RESERVED\_PIDS;

if (unlikely(last == offset))

break;

}

pid = mk\_pid(pid\_ns, map, offset);

}

return -1;

}

pid允许分配的最大值为32767，当pid分配轮过一圈之后则允许分配的最小值为300，也就是说前300个pid是不可再分配的。

相关常量如下：

#define PAGE\_SHIFT 12

#define PAGE\_SIZE (1UL << PAGE\_SHIFT) // 2^12

#define BITS\_PER\_PAGE (PAGE\_SIZE \* 8) // 2^15

#define BITS\_PER\_PAGE\_MASK (BITS\_PER\_PAGE-1) //2^15-1

#define PAGE\_MASK (~(PAGE\_SIZE-1))

#### 2.3.1 pidmap结构体

[-> kernel/include/linux/pid\_namespace.h]

struct pidmap {

atomic\_t nr\_free; //可用pid的个数

void \*page; //用于存放位图

};

pidmap->page的大小为4KB，每一个bit位代表一个进程pid的分配情况，那么4KB\*8=32768， 这正好是pid可分配的上限，用nr\_free代表该namespace下还有多少可用pid。

#### 2.3.2 find\_next\_offset

[-> pid.c]

#define find\_next\_offset(map, off) \ find\_next\_zero\_bit((map)->page, BITS\_PER\_PAGE, off)

static inline int mk\_pid(struct pid\_namespace \*pid\_ns, struct pidmap \*map, int off) {

return (map - pid\_ns->pidmap)\*BITS\_PER\_PAGE + off;

}

### 2.4 pid\_nr

[-> kernel/include/linux/pid.h]

static inline pid\_t pid\_nr(struct pid \*pid) {

pid\_t nr = 0;

if (pid)

nr = pid->numbers[0].nr;

return nr;

}

根据pid结构体找到真正的pid数值。

## 三. 总结

* pid分配上限的查询方式cat /proc/sys/kernel/pid\_max，Android系统一般默认为32768。
* 对于pid<300的情况值允许分配一次，不可再改变。也就是进程pid分配范围为(300, 32768)；
* 每个pid分配成功，便会把当前的pid设置到last\_pid， 那么下次pid的分配便是从last\_pid+1开始 往下查找。这就意味着当last\_pid+1或者附近的进程，刚被杀并回收该pid，此时再创建新进程，很有可能会复用 pid.
* 位图法记录已分配和未分配pid,由于pid的最大上限为32768，故pidmap采用4KB大小的内存，每一位代表一个进程ID号，正好4K\*8=32K= 32768

# /proc/stat解析

*本文解析进程节点含义*

**一. 概述**

**1.1 CPU时间**

| **cpu指标** | **含义** |
| --- | --- |
| user | 用户态时间 |
| nice | 用户态时间(低优先级，nice>0) |
| system | 内核态时间 |
| idle | 空闲时间 |
| iowait | I/O等待时间 |
| irq | 硬中断 |
| softirq | 软中断 |

iowait时间是不可靠值，理由如下：

* CPU不会等待I/O执行完成，而iowait是等待I/O完成的时间。 当CPU进入idle状态，很可能会调度另一个task执行，所以iowait计算时间偏小；
* 多核CPU，iowait的计算并非某一个核，因此计算每一个cpu的iowait非常困难；

相关资料：http://man7.org/linux/man-pages/man5/proc.5.html

**二. proc/stat**

proc/stat节点记录的是系统进程整体的统计信息

**2.1 stat数据**

Gityuan$ adb shell cat /proc/stat

//CPU指标：user，nice, system, idle, iowait, irq, softirq

cpu 130216 19944 162525 1491240 3784 24749 17773 0 0 0

cpu0 40321 11452 49784 403099 2615 6076 6748 0 0 0

cpu1 26585 2425 36639 151166 404 2533 3541 0 0 0

cpu2 22555 2957 31482 152460 330 2236 2473 0 0 0

cpu3 15232 1243 20945 153740 303 1985 3432 0 0 0

cpu4 5903 595 6017 157410 30 10959 605 0 0 0

cpu5 4716 380 3794 157909 23 118 181 0 0 0

cpu6 8001 515 8995 157571 48 571 180 0 0 0

cpu7 6903 377 4869 157885 31 271 613 0 0 0

intr ...

ctxt 22523049

btime 1500827856

processes 23231

procs\_running 1

procs\_blocked 0

softirq 3552900 843593 733695 19691 93143 468832 12783 257382 610426 0 513355

时间单位，sysconf(\_SC\_CLK\_TCK)一般地定义为jiffies(一般地等于10ms)

**2.2 说明**

1. intr：系统启动以来的所有interrupts的次数情况
2. ctxt: 系统上下文切换次数
3. btime：启动时长(单位:秒)，从Epoch(即1970零时)开始到系统启动所经过的时长，每次启动会改变。
   * 此处指为1500827856，转换北京时间为2017/7/24 0:37:36
4. processes：系统启动后所创建过的进程数量。当短时间该值特别大，系统可能出现异常
5. procs\_running：处于Runnable状态的进程个数
6. procs\_blocked：处于等待I/O完成的进程个数

另外：

cat /proc/uptime

82044.14 215440.94

* 第一个值代表从开机到现在的累积时间(单位为秒), 开机后运行82044秒
* 第二个值代表从开机到现在的CPU空闲时间，单位为秒

技巧：结合btime获取当前的绝对时间,1500827856 + 82044 = 1500909900， 转换成北京时间2017/7/24 23:25:00，也就是当前执行命令cat /proc/uptime的时间点。

**三. proc/[pid]>/stat**

proc//stat用于获取某一个进程的统计信息，实现过程见fs/proc/array.c的do\_task\_stat()

**3.1 stat数据**

Gityuan$ adb shell cat /proc/8385/stat

1557 (system\_server) S 823 823 0 0 -1 1077952832 //1~9

2085481 15248 2003 27 166114 129684 26 30 //10~17

10 -10 221 0 2284 2790821888 93087 18446744073709551615 //18~25

1 1 0 0 0 0 6660 0 36088 0 0 0 17 3 0 0 0 0 0 0 0 0 0 0 0 0 0

**3.2 解释**

1. pid： 进程ID.
2. comm: task\_struct结构体的进程名
3. state: 进程状态, 此处为S
4. ppid: 父进程ID （父进程是指通过fork方式，通过clone并非父进程）
5. pgrp：进程组ID
6. session：进程会话组ID
7. tty\_nr：当前进程的tty终点设备号
8. tpgid：控制进程终端的前台进程号
9. flags：进程标识位，定义在include/linux/sched.h中的PF\_\*, 此处等于1077952832
10. minflt： 次要缺页中断的次数，即无需从磁盘加载内存页. 比如COW和匿名页
11. cminflt：当前进程等待子进程的minflt
12. majflt：主要缺页中断的次数，需要从磁盘加载内存页. 比如map文件
13. majflt：当前进程等待子进程的majflt
14. utime: 该进程处于用户态的时间，单位jiffies，此处等于166114
15. stime: 该进程处于内核态的时间，单位jiffies，此处等于129684
16. cutime：当前进程等待子进程的utime
17. cstime: 当前进程等待子进程的utime
18. priority: 进程优先级, 此次等于10.
19. nice: nice值，取值范围[19, -20]，此处等于-10
20. num\_threads: 线程个数, 此处等于221
21. itrealvalue: 该字段已废弃，恒等于0
22. starttime：自系统启动后的进程创建时间，单位jiffies，此处等于2284
23. vsize：进程的虚拟内存大小，单位为bytes
24. rss: 进程独占内存+共享库，单位pages，此处等于93087
25. rsslim: rss大小上限

说明：

* 第10~17行主要是随着时间而改变的量；
* 内核时间单位，sysconf(\_SC\_CLK\_TCK)一般地定义为jiffies(一般地等于10ms)
* starttime: 此值单位为jiffies, 结合/proc/stat的btime，可知道每一个线程启动的时间点
  + 1500827856 + 2284/100 = 1500827856, 转换成北京时间为2017/7/24 0:37:58

第四行数据很少使用,只说一下该行第7至9个数的含义:

* signal：即将要处理的信号，十进制，此处等于6660
* blocked：阻塞的信号，十进制
* sigignore：被忽略的信号，十进制，此处等于36088

# 参考

<http://gityuan.com/2017/07/30/linux-process/>

<http://gityuan.com/2017/08/05/linux-process-fork/>

http://gityuan.com/2017/08/06/linux\_process\_pid/