Linux 源码分析 系列之

进程

张文力
zhangwl@ncic.ac.cn
智能中心 ASL

2004.3

讲述提纲

- 概述
- 相关数据结构 （task_struct）
- 进程的调度
- 进程的控制

概述

- 相关概念
- 进程在整个内核中的功能位置
- 源代码中进程相关的文件
- Linux进程的四个要素

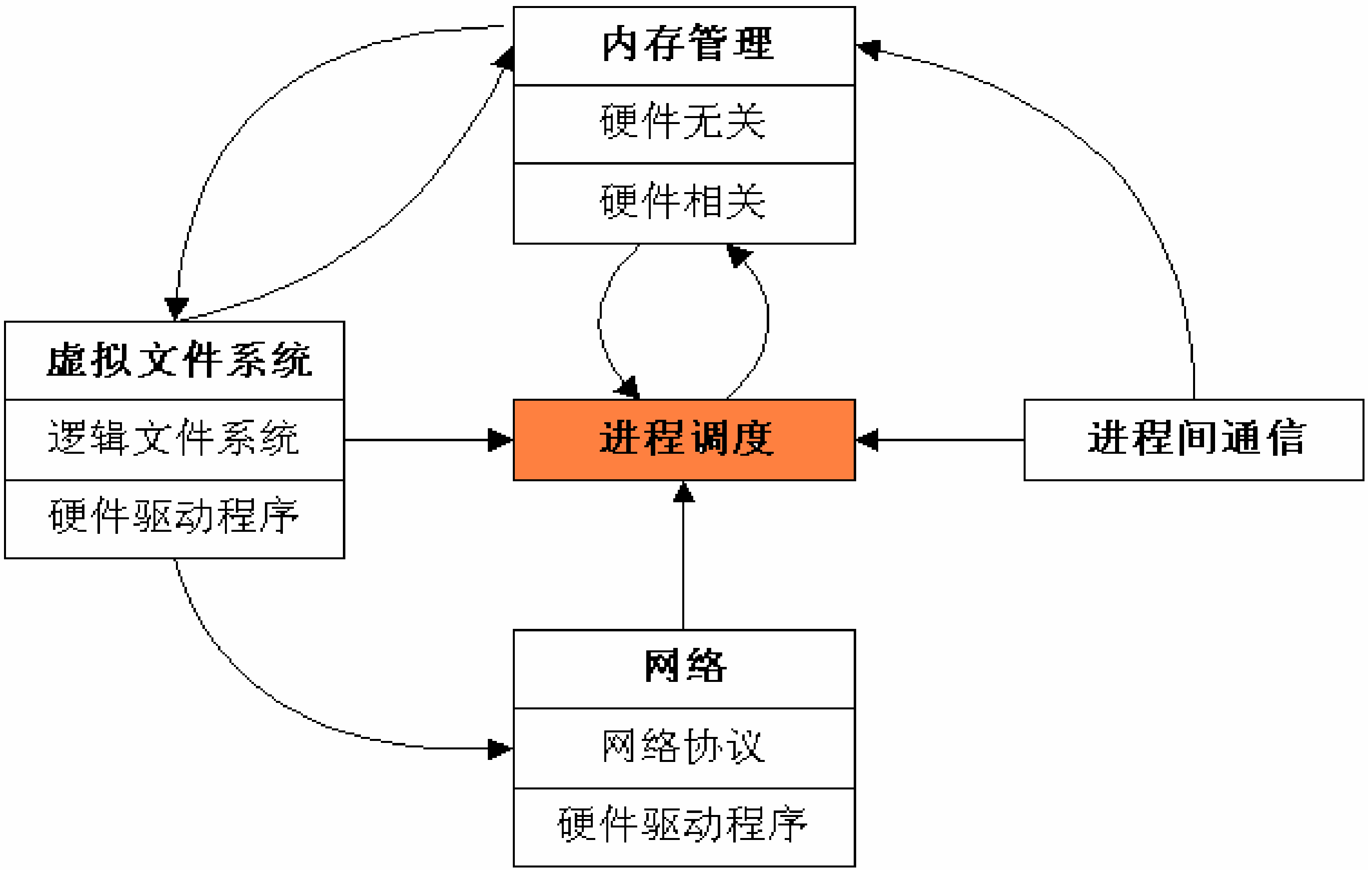
相关概念

- Linux核心是多任务的
- 运行的程序称作进程 (process)
- 线程 (Thread) 为单一进程提供了做多件事的可能
  - Threads within a thread group share all their global variables and have the same heap. But the threads have **different stacks** (they don’t share local variable)
  - Threads are processes that happen to share the global memory space.
- Linux内核中赋予二者更通用的名称——任务 (*task*)

相关概念

- 调度 scheduling –对CPU访问的仲裁
  - – divides time into “slices”
  - – allocates the slices to processes according to some algorithm
- 调度算法 –软件实时 soft realtime
- 优先级 – Static(priority), Dynamic(counter), Realtime(rt_priority)
- 进程ID (PIDs)
  - – Every Linux process has a unique identifier which is called its process identifier (PID)
- 引用计数 Reference Counting
  - – In general terms, one or more “holder” objects carry a pointer to a shared data object that includes an integer called its reference count
- 权能 weight –精确的定义经授权的进程所允许处理的事情

进程在整个内核中的功能位置



源代码中进程相关的文件

- include/linux/
  - sched.h *
  - interrupt.h:
  - tqueue.h
  - wait.h:
  - locks.h
  - spinlock.h
  - smp_lock.h
  - timer.h
  - signal.h
- kernel/
  - sched.c
  - softirq.c
  - fork.c
  - itimer.c
  - signal.c
- ipc/
  - msg.c
  - shm.c
  - sem.c
  - util.c
- arch/i386/kernel
  - process.c
  - entry.s
  - traps.c
  - irq.c, irq.h
- include/asm-i386/
  - processor.h
  - irq.h, hardirq.h, softirq.h
  - semaphore.h
  - locks.h, spinlock.h, smplock.h
  - ......

Linux进程的四个要素

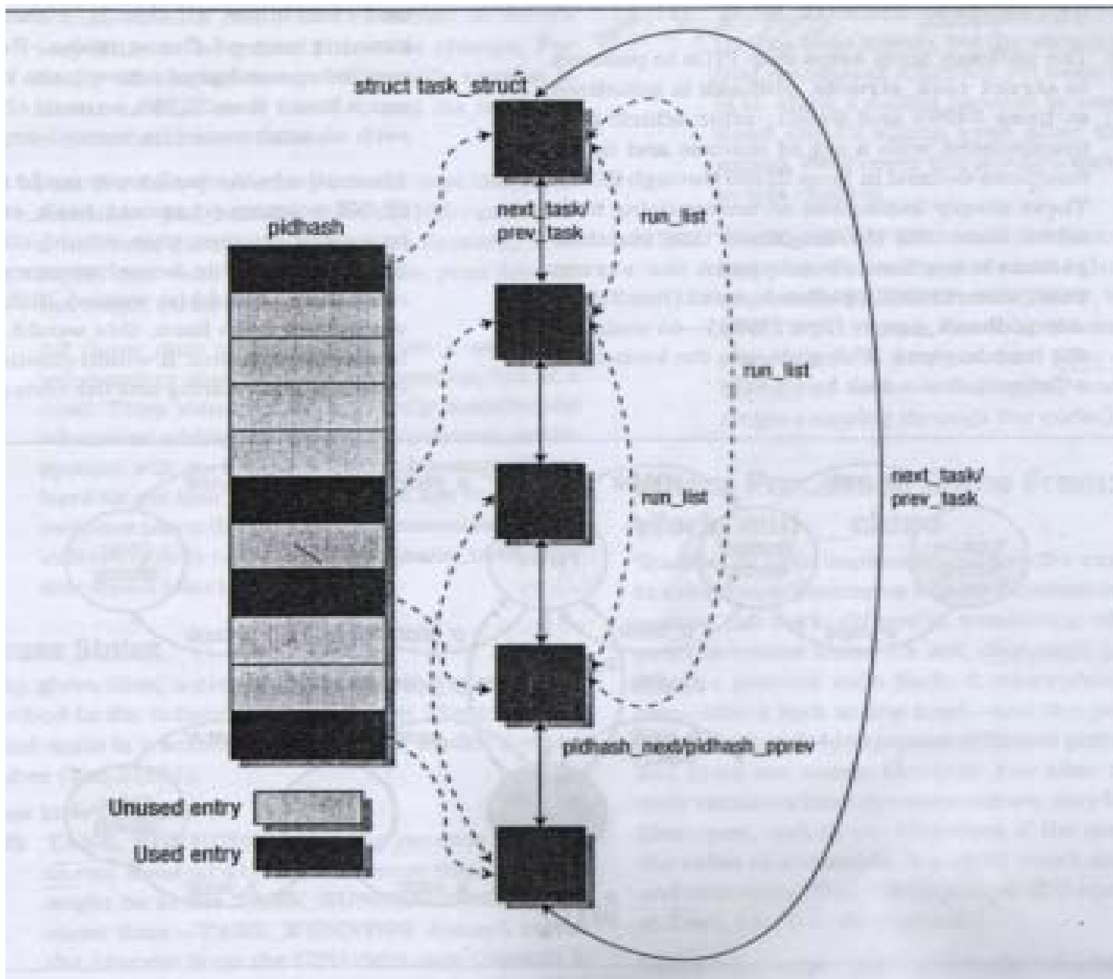
- 有一段程序供其执行。这段程序不一定是某个进程所专有，可以与其他进程共用。
- 有进程专用的内核空间堆栈。
- 在内核中有一个task_struct数据结构，即通常所说的“进程控制块”。有了这个数据结构，进程才能成为内核调度的一个基本单位接受内核的调度。同时，这个结构还记录着进程所占用的各项资源。
- 有独立的存储空间，这意味着拥有专有的用户空间；进一步，还意味着除前述的内核空间堆栈外还有其专用的用户空间堆栈。有一点必须指出，内核空间是不能独立的，任何进程都不可能直接（不通过系统调用）改变内核空间的内容（除其本身的内核空间堆栈以外）。

相关数据结构

- 进程在内核中的表示
- Struct task_struct
- Process ID (PID)

进程在内核中的表示

- 每个进程以一个 *task_struct* 数据结构表示
- 几个相关的用于跟踪进程的核心数据结构
  - Task-list
  - Run-queue
  - Parent-child-relationship
  - PID hash table



Struct task_struct

- ***include/linux/sched.h***
  - 调度数据成员 states, flags, priority…
  - 信号处理 signal …
  - 进程队列指针 *next_task，*prev_task …
  - 进程标识 pid, uid，gid …
  - 时间数据成员 timeout …
  - 信号量数据成员 *semsleeping …
  - 进程上下文环境 tss, saved_kernel_stack …
  - 文件系统数据成员 *fs, *files …
  - 内存数据成员 *mm …
  - 页面管理 swap_address, nswap …
  - 支持对称多处理器方式(SMP)时的数据成员 processor, lock_depth …
  - 其它数据成员 rlim[RLIM_NLIMITS] …
  - 进程队列的全局变量 current, *task[NR_TASKS] …

Struct task_struct

/include/linux/sched

```
struct task_struct{
    volatile long state;
    ...
    struct list_head run_list;
    ...
    struct task_struct *next_task, *p
    ...
    pid_t pid;
    ...
    struct task_struct *p_opptr, *p_
    ...
    struct task_struct *pidhash_next
    struct task_struct **pidhash_ppi
    ...
}
```

/include/linux/sched.h

#define TASK_RUNNING 0

-正在运行或在就绪队列run-queue中准备运行的进程，实际参与进程调度。

#define TASK_INTERRUPTIBLE 1

-处于等待队列中的进程，待资源有效时唤醒，也可由其它进程通过信号(signal)或定时中断唤醒后进入就绪队列run-queue。

#define TASK_UNINTERRUPTIBLE 2

-处于等待队列中的进程，待资源有效时唤醒，不可由其它进程通过信号(signal)或定时中断唤醒。

#define TASK_ZOMBIE 4

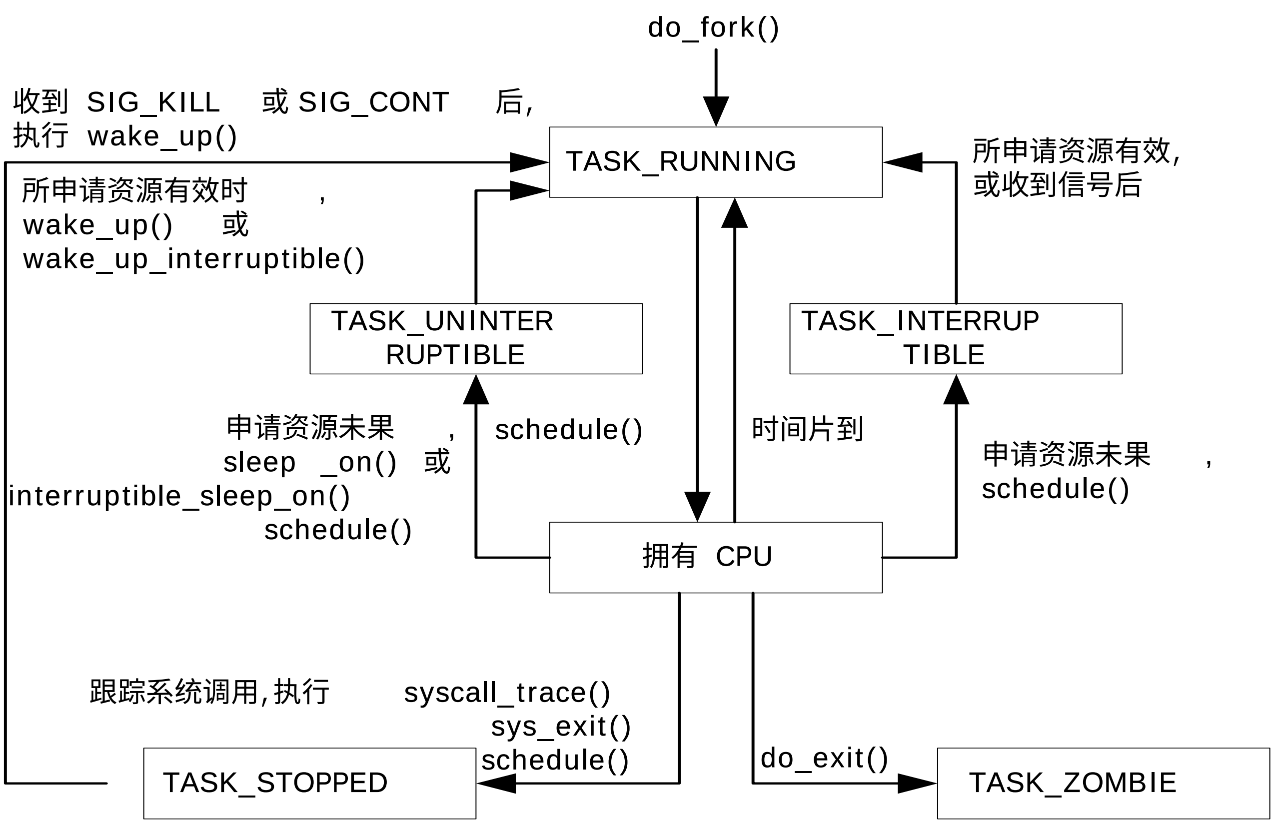
-表示进程结束但尚未消亡的一种状态(僵死状态)。此时，进程已经结束运行且释放大部分资源，但尚未释放进程控制块。

#define TASK_STOPPED 8

-进程被暂停，通过其它进程的信号才能唤醒。导致这种状态的原因有二，或者是对收到SIGSTOP、SIGSTP、SIGTTIN或SIGTTOU信号的反应，或者是受其它进程的ptrace系统调用的控制而暂时将CPU交给控制进程。

#define TASK_SWAPPING 16

-主要用于表明进程正在执行磁盘交换工作。但其值从来没有被赋给进程的**state**成员。这种状态已被淘汰。



Linux 进程的状态转换

Struct task_struct

/include/linux/sched

```
struct task_struct{
    volatile long state;
    ...
    struct list_head run_list;
    ...
    struct task_struct *next_task, *p
    ...
    pid_t pid;
    ...
    struct task_struct *p_opptr, *p_
    ...
    struct task_struct *pidhash_next;
    struct task_struct **pidhash_pprev;
    ...
}
```

/include/linux/list.h

```
struct list_head {
        struct list_head *next, *prev;
};
#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)
```

/include/linux/sched.h

```
#define INIT_TASK(tsk) \
{
    …
    Run_list: LIST_HEAD_INIT(tsk.run_list), \
    …
}
```

/include/linux/sched.c

```
static LIST_HEAD(runqueue_head);
```

Struct task_struct

/include/linux/sched

```
struct task_struct{
    volatile long state;
    ...
    struct list_head run_list;
    ...
    struct task_struct *next_task, *p
    ...
    pid_t pid;
    ...
    struct task_struct *p_opptr, *p_
    ...
    struct task_struct *pidhash_next
    struct task_struct **pidhash_pp
    ...
}
```

/include/linux/list.h

```
static inline void add_to_runqueue(struct task_struct * p)
{
        list_add_tail(&p->run_list, &runqueue_head);
        nr_running++;
}

static inline void move_last_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add_tail(&p->run_list, &runqueue_head);
}
```

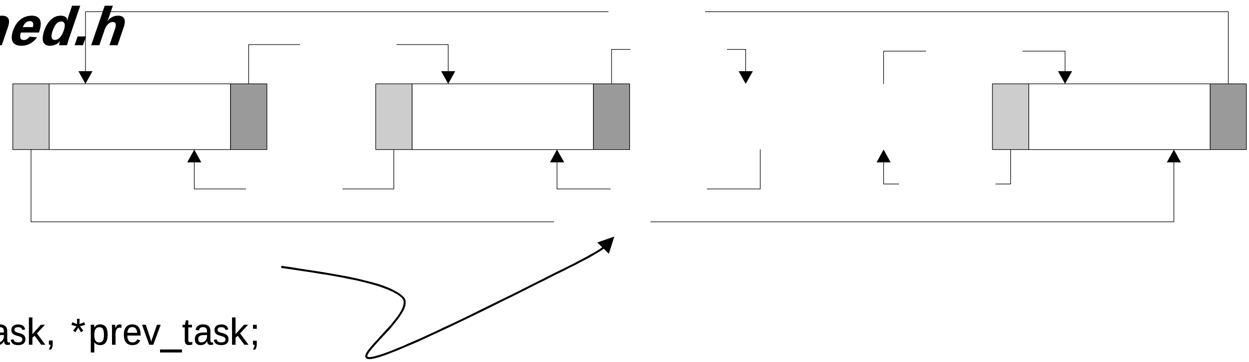
/include/linux/sched.h

```
static inline void del_from_runqueue(struct task_struct * p)
{
        nr_running--;
        p->sleep_time = jiffies;
        list_del(&p->run_list);
        p->run_list.next = NULL;
}

static inline int  task_on_runqueue(struct task_struct *p)
{
        return (p->run_list.next != NULL);
}
```

Struct task_struct

/include/linux/sched.h

```
struct task_struct{
      volatile long state;
      ...
      struct list_head run_list;
      ...
      struct task_struct *next_task, *prev_task;
      ...
      pid_t pid;
      ...
      struct task_struct *p_opptr, *p_pptr,
      *p_cptr, *p_ysptr, *p_osptr;
      ...
      struct task_struct *pidhash_next;
      struct task_struct **pidhash_pprev;
      ...
}
```



Init_task

Struct task_struct

/include/linux/sched.h

```
struct task_struct{
    volatile long state;
    ...
    struct list_head run_list;
    ...
    struct task_struct *next_task, *prev_task;
    ...
    pid_t pid;
    ...
    struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
    ...
    struct task_struct *pidhash_next;
    struct task_struct **pidhash_pprev;
    ...
}
```

/include/linux/types.h

typedef __kernel_pid_t pid_t;

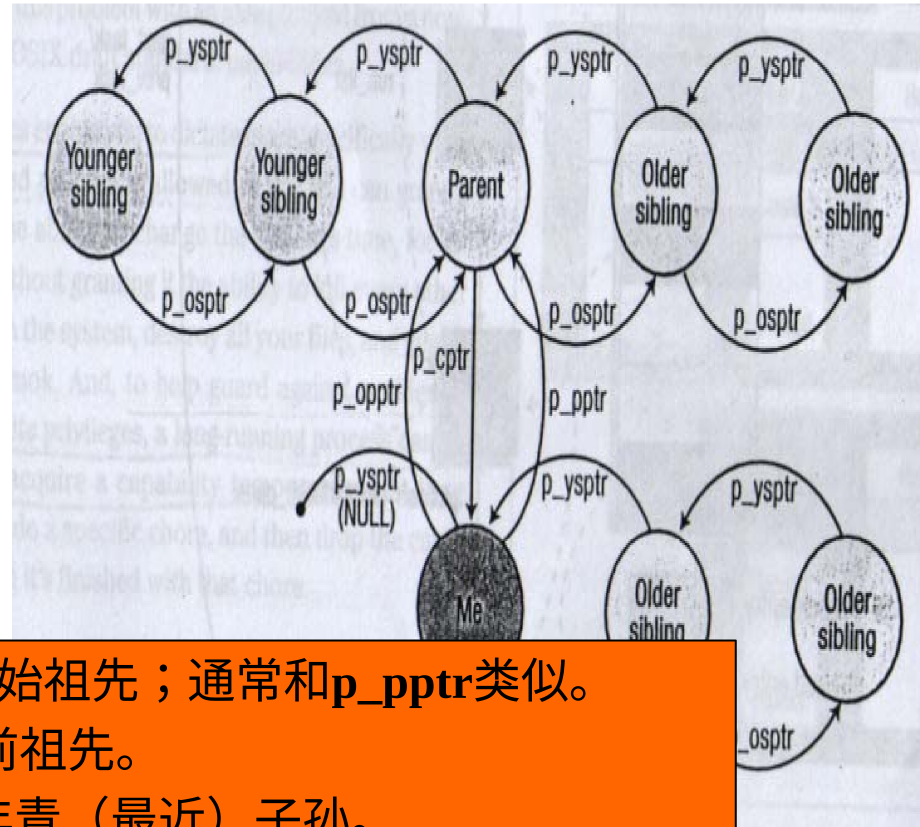
/include/linux/asm-386/Posix_types.h

typedef int __kernel_pid_t;

Struct task_struct

/include/linux/sched.h

```
struct task_struct{
        volatile long state;
        ...
        struct list_head run_list;
        ...
        struct task_struct *next_task, *prev_task;
        ...
        pid_t pid;
        ...
        struct task_struct *p_opptr, *p_pptr,
        *p_cptr, *p_ysptr, *p_osptr;
        ...
        struct task_
        struct task_
        ...
}
```



1) **p_opptr**指向进程的原始祖先；通常和**p_pptr**类似。
2) **p_pptr**指向进程的当前祖先。
3) **p_cptr**指向进程的最年青（最近）子孙。
4) **p_ysptr**指向进程的下一个最年青（下一个最近）兄弟。
5) **p_osptr**指向进程的下一个最古老（下一个最远）兄弟。

Struct task_struct

/include/linux/sched.h

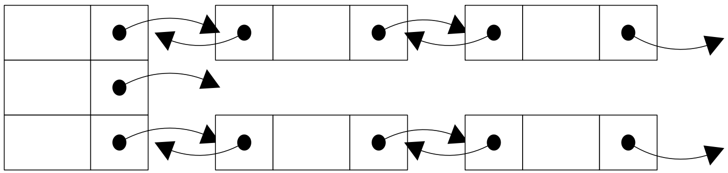
```
struct task_struct{
    volatile long state;
    ...
    struct list_head run_list;
    ...
    struct task_struct *next_task, *prev_task;
    ...
    pid_t pid;
    ...
    struct task_struct *p_opptr, *p_pptr, *p
    ...
    struct task_struct *pidhash_next;
    struct task_struct **pidhash_pprev;
    ...
}
```

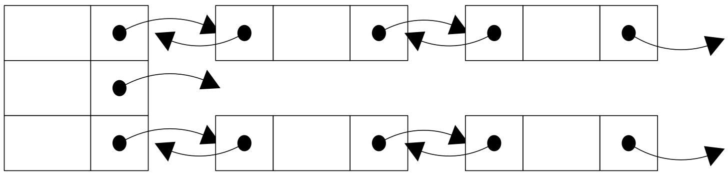
后面详细说明

Process ID (PID)

/include/linux/sched.h

```
#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];
#define pid_hashfn(x)  ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))

static inline void hash_pid(struct task_struct *p)
{
        struct task_struct **htable = &pidhash[pid_hashfn(p->pid)];

        if((p->pidhash_next = *htable) != NULL)
                (*htable)->pidhash_pprev = &p->pidhash_next;
        *htable = p;
        p->pidhash_pprev = htable;
}

static inline void unhash_pid(struct task_struct *p)
{
        if(p->pidhash_next)
                p->pidhash_next->pidhash_pprev = p->pidhash_pprev;
        *p->pidhash_pprev = p->pidhash_next;
}

static inline struct task_struct *find_task_by_pid(int pid)
{
        struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];
        for(p = *htable; p && p->pid != pid; p = p->pidhash_next)
                ;
        return p;
}
```





Process ID (PID)

/kernel/fork.c

```
struct task_struct *pidhash[PIDHASH_SZ];
int last_pid;
static int get_pid(unsigned long flags)
{
    static int next_safe = PID_MAX;
    struct task_struct *p;
    int pid, beginpid;

    if (flags & CLONE_PID)
        return current->pid;

    spin_lock(&lastpid_lock);
    beginpid = last_pid;
    if((++last_pid) & 0xffff8000) {
        last_pid = 300;
        goto inside;
    }
```

The **next_safe** variable is a speed hack; it keeps track of the next-lowest candidate PID that might be reserved

/include/linux/Threads.h

#define PID_MAX 0x8000

The **last_pid** variable is the PID that was taken by last task

位与运算只是通过测试低15位是否置位来简单测试**last_pid**的新值是否超过了32,767（最大允许的PID）。
如果**last_pid** 超过最大允许值, 回滚到300。而300对于内核并没有特别的意义。

Process ID (PID)

```
        if(last_pid >= next_safe) {
inside:
                next_safe = PID_MAX;
                read_lock(&tasklist_lock);
        repeat:
                for_each_task(p) {
                        if(p->pid == last_pid        ||
                           p->pgrp == last_pid       ||
                           p->tgid == last_pid       ||
                           p->session == last_pid) {
                                if(++last_pid >= next_safe) {
                                        if(last_pid & 0xffff8000)
                                                last_pid = 300;
                                        next_safe = PID_MAX;
                                }
                                if(unlikely(last_pid == beginpid))
                                        goto nomorepids;
                                goto repeat;
                        }
                                ...
                }
                read_unlock(&tasklist_lock);
        }
        pid = last_pid;
        spin_unlock(&lastpid_lock);

        return pid;

...
}
```

/include/linux/Sched.h

```
#define for_each_task(p) \
   for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

If a new task is being created and the only available PIDs are below 300, this loop will simply continue until some process with a higher-numbered PID exits.

进程的调度

- 进程调度的策略
- 进程的调度算法
- Linux进程调度的全过程

进程调度的策略

- 进程调度的策略主要考虑以下几个原则：
  - 高效 使处理器的利用率最高，空闲最小；
  - 公平 使每一个申请处理器的进程都得到合理的处理器时间；
  - 周转时间短 使用户提交任务后得到结果的时间尽可能短；
  - 吞吐量大 使单位时间内处理的任务数量尽可能多；
  - 响应时间短 使对每个用户响应的时间尽可能短。
- 进程调度的策略：
  - 实时进程有两种策略：
    - 时间片轮转(round robin) 依次运行
    - 先进先出（first in first out）按照在调度队列中的顺序运行，这个顺序不会改变。
  - 非实时进程，Linux永远选择优先级最高的进程来运行。

进程的调度算法

- 时间片轮转调度算法
- 优先级调度算法
  - 用于非实时进程。Linux采用抢占式的优先级算法，即系统中当前运行的进程永远是可运行进程中优先权最高的那个。
- 先进先出调度算法
  - 采用FIFO的实时进程必须是运行时间较短的进程，因为这种进程一旦获得CPU就只有等到它运行完或因等待资源主动放弃CPU时其它进程才能获得运行机会。（见）

```
/ kernel/sched.h
#define SCHED_OTHER          0
#define SCHED_FIFO           1
#define SCHED_RR             2
```

Linux进程调度的全过程 1/2

⌘ 转入schedule处理函数前的过程

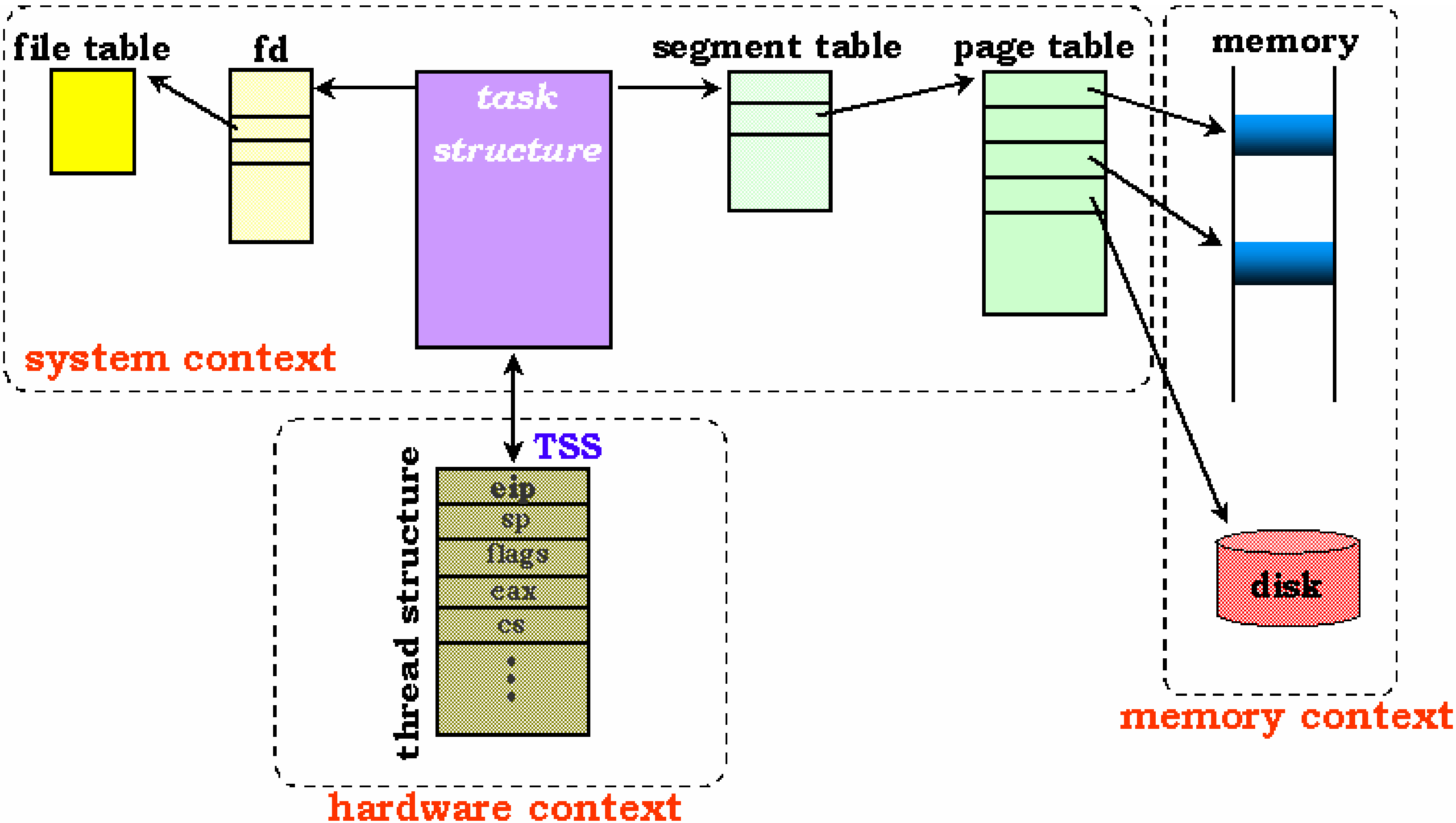
–根据调度时机以五种方式转入schedule处理函数

- 进程状态转换时
  - -当进程要调用sleep（）或exit（）等函数
- 通过时钟中断
  - -周期性时间中断, 因此至多每隔10ms的周期，进程管理器(函数 schedule())就会被调用。
- 内核处理完中断服务，返回到用户态时
  - -系统会返回到入口ret_from_intr,再跳转到ret_with_reschedule，判断need_resched标志是否置位，若是则转入执行schedule()。
- 进程从系统调用返回到用户态时
  - -到ret_with_reschedule，判断need_resched标志是否置位，若是则转入执行schedule()。
- 运行队列增加进程时
  - -add_to_runqueue() 中比较要加入的进程和正在运行的进程的counter值

Linux进程调度的全过程 2/2

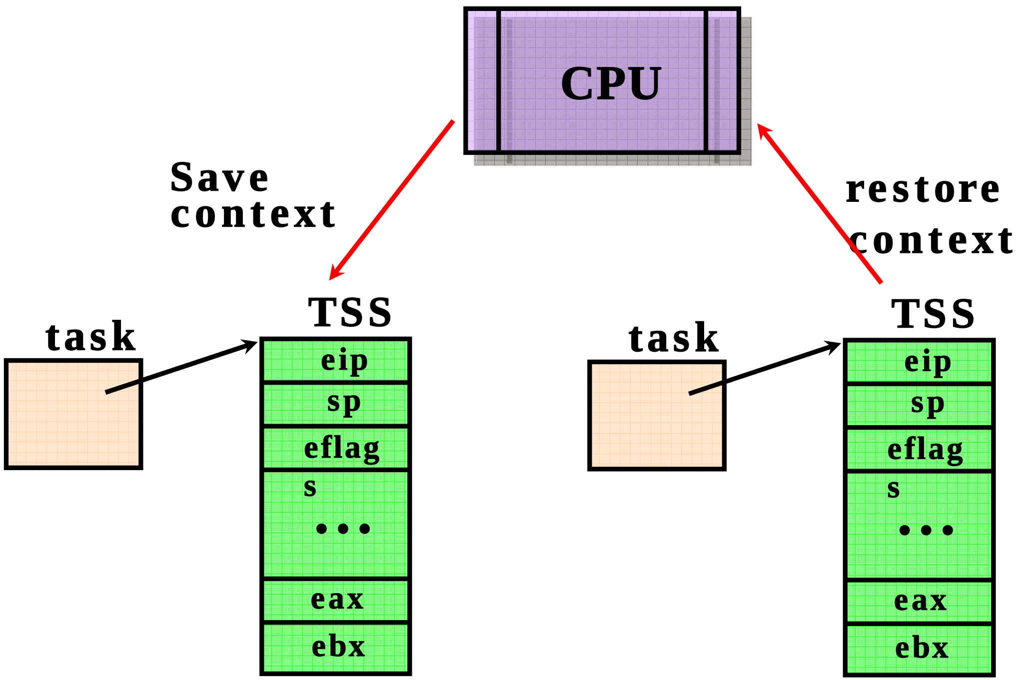
- 执行schedule( ),完成进程调度，切换进程
  - 在schedule中，先检查是否是中断服务程序调用了schedule（这是不允许的），如果是则退出schedule。若不是，则检查是否有bottom half服务请求，若有则执行do_half_bottom。
  - 然后再判断当前进程是否是采用RR调度法的实时进程，若是则重新给它分配时间片并把它移到运行队列尾部。接着根据当前进程的状态对当前进程作相关处理。
  - 接下来，便是调度正文。通过函数goodness( )遍历运行队列中所有的进程，选择权值最大的进程作为下一个运行的进程。若运行队列中所有的进程的时间片都耗尽了，则要给系统中所有的进程重新分配时间片。
  - 最后通过宏switch_to()切换堆栈，从而达到从当前进程切换到选中的进程的目的。调度结束。
- 本部分主要涉及schedule( )、goodness( )两个函数

上下文信息



file table
fd
task structure
segment table
page table
memory
system context
TSS
thread structure
eip
sp
flags
eax
cs
disk
memory context
hardware context

Hardware context switch




```
/* need context switch */
if (save_context())
{
/* pick another task to run from run_queue */
    ….
restore_context(new task)
/* The control does not arrive here, NEVER !!! */
}
/* resuming process executes from here !!! */
……
```

进程的控制

- 子进程的创建
- 子进程的执行
- 父进程的等待
- 子进程的消亡
- 进程控制的全过程

子进程的创建

- 子进程的创建：fork( )、clone( )和vfork( )
- 这三个函数都是通过调用do_fork( )来实现具体创建工作的，但它们传递给do_fork( )的参数不同，因此创建出的进程是不同的。
- sys_clone( )用于创建一个线程，这个线程可以是内核线程，也可以是用户线程。
- sys_fork( )用于全面地复制，创建出的新进程与父进程几乎相同。
- sys_vfork( )也用于创建线程。但主要只是作为创建进程的中间步骤，目的在于提高创建时的效率，减少系统开销。

子进程的创建

/arch/i386/kernel/process.c

```
asmlinkage int sys_fork(struct pt_regs regs)
{
	return do_fork(SIGCHLD, regs.esp, &regs, 0);
}

asmlinkage int sys_clone(struct pt_regs regs)
{
	unsigned long clone_flags;
	unsigned long newsp;

	clone_flags = regs.ebx;
	newsp = regs.ecx;
	if (!newsp)
		newsp = regs.esp;
	return do_fork(clone_flags, newsp, &regs, 0);
}
asmlinkage int sys_vfork(struct pt_regs regs)
{
	return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, &regs, 0);
}
```

do_fork()函数流程

1)新建子进程分配task_struct 空间
2)子进程task_struct的初始化
3)子进程file,fs,sighand,mm,
  thread五方面信息的复制
4)出错滚回处理[详见exit_sighand()]

子进程的创建

/kernel/fork.c

```
int do_fork(unsigned long clone_flags, unsigned long stack_start,
            struct pt_regs *regs, unsigned long stack_size)
{
        int retval;
        struct task_struct *p;
        struct completion vfork;

        if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) =      NE_NEWNS|CLONE_FS))
                return -EINVAL;

        retval = -EPERM;

        if (clone_flags &
                if (cu

        }

        retval = -ENOMEM;
        p = alloc_task_struct();
        if (!p)
                goto fork_out;

        *p = *current;
```

CLONE_PID is only allowed for the initial SMP swapper calls

新建子进程分配task_struct 空间

/include/asm-i386/Processor.h

```
#define alloc_task_struct() ((struct task_struct *) __get_free_pages(GFP_KERNEL,1))
```

子进程的创建

The current, running, process is pointed to by the current pointer.

/include/asm-i386/Current.h

```
static inline struct task_struct * get_current(void)
{
  struct task_struct *current;
  __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
  return current;
 }
#define current get_current()
```

/kernel/fork.c

```
int do_fork(unsigned long clone_flags, unsign
        struct pt_regs *regs, unsigned long s
{
        int retval;
        struct task_struct *p;
        struct completion vfork;

        if ((clone_flags & (CLONE_NEW
                return -EINVAL;

        retval = -EPERM;

        if (clone_flags & CLONE_PID) {
                if (current->pid)
                                goto fork_out;
        }

        retval = -ENOMEM;
        p = alloc_task_struct();
        if (!p)
                goto fork_out;

        *p = *current;
```

子进程的创建

```
        p->pid = get_pid(clone_flags);
...
p->run_list.next = NULL;
p->run_list.prev = NULL;
...
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & CLONE_PARENT)) {
        p->p_opptr = current;
        if (!(p->ptrace & PT_PTRACED))
          p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
        p->tgid = current->tgid;
        list_add(&p->thread_group, &current->threa
}
...
SET_LINKS(p);
hash_pid(p);
nr_threads++;
...
wake_up_process(p);                    /* do this last */
++total_forks;
if (clone_flags & CLONE_VFORK)
        wait_for_completion(&vfork);
```

Get a PID belong to this task

子进程task_struct的初始化

Initially, the new process is not placed in the run queue. It's still possible that do_fork will fail, and it would be wasteful to put the new process in the run queue, only to take it right back out if something goes wrong. More seriously, the new task is not completely initialized yet, and we wouldn't want another CPU to hand control to this process prematurely.

子进程的创建

```
p->pid = get_pid(clone_flags);
...
p->run_list.next = NULL;
p->run_list.prev = NULL;
...
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & CLONE_PARENT)) {
        p->p_opptr = current;
        if (!(p->ptrace & PT_PTRACED))
          p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
        p->tgid = current->tgid;
        list_add(&p->thread_group, &current->thread_
}
...
SET_LINKS(p);
hash_pid(p);
nr_threads++;
...
wake_up_process(p);                /* do this last */
++total_forks;
if (clone_flags & CLONE_VFORK)
        wait_for_completion(&vfork);
```

Normally, do_fork's caller should be registered as the parent of the new process. The only exception is when the CLONE_PARENT flag is set. This flag means that the new process should have the same parent as do_fork's caller.

If the calling process is not being traced, the caller is also made the new process's logical parent – the one to which signal notifications are sent. However, if the caller is being traced, the child's logical parent will remain the same as its parent's logical parent, the debugger process.

子进程的创建

```
p->pid = get_pid(clone_flags);
...
p->run_list.next = NULL;
p->run_list.prev = NULL;
...
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & CLONE_PARENT)) {
	p->p_opptr = current;
	if (!(p->ptrace & PT_PTRACED))
	  p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
	p->tgid = current->tgid;
	list_add(&p->thread_group, &curre
}
...
SET_LINKS(p);
hash_pid(p);
...
wake_up_process(p);
++total_forks;
if (clone_flags & CLONE_VFORK)
	wait_for_completion(&vfork);
```

/include/linux/Sched.h

```
#define SET_LINKS(p) do { \
	(p)->next_task = &init_task; \ 进程p的下一个进程是初始化进程
	(p)->prev_task = init_task.prev_task; \ p的前一个进程是初始化进程的前一个
	init_task.prev_task->next_task = (p); \ 进程p的进一进程指向p
	init_task.prev_task = (p); \  初始化进程的前一进程指向p
	; 即将进程p加入到环形进程队列的尾部
	(p)->p_ysptr = NULL; \ 进程p现在是最年轻的进程
	if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL) \
		(p)->p_osptr->p_ysptr = p; \ 原来的最年轻进程变成p的兄进程
	(p)->p_pptr->p_cptr = p; \ 父进程指向新的子进程p
	} while (0)
```

子进程的创建

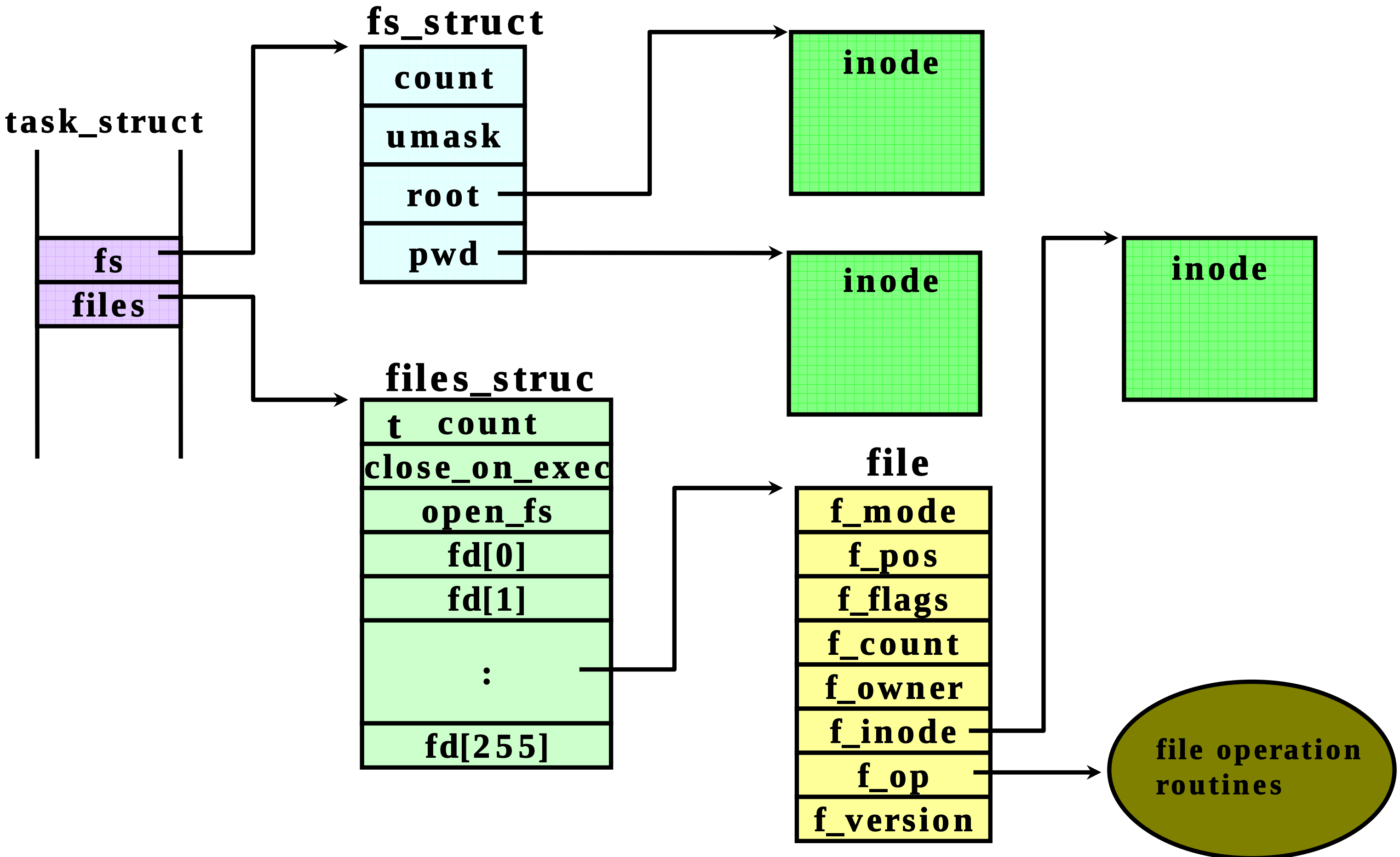
```
p->pid = get_pid(clone_flags);
...
p->run_list.next = NULL;
p->run_list.prev = NULL;
...
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & CLONE_PARENT)) {
        p->p_opptr = current;
        if (!(p->ptrace & PT_PTRACED))
          p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
        p->tgid = current->tgid;
        list_add(&p->thread_group, &curre
}
...
SET_LINKS(p);
hash_pid(p);
...
wake_up_process(p);
++total_forks;
if (clone_flags & CLONE_VFORK)
        wait_for_completion(&vfork);
```

/include/linux/Sched.h

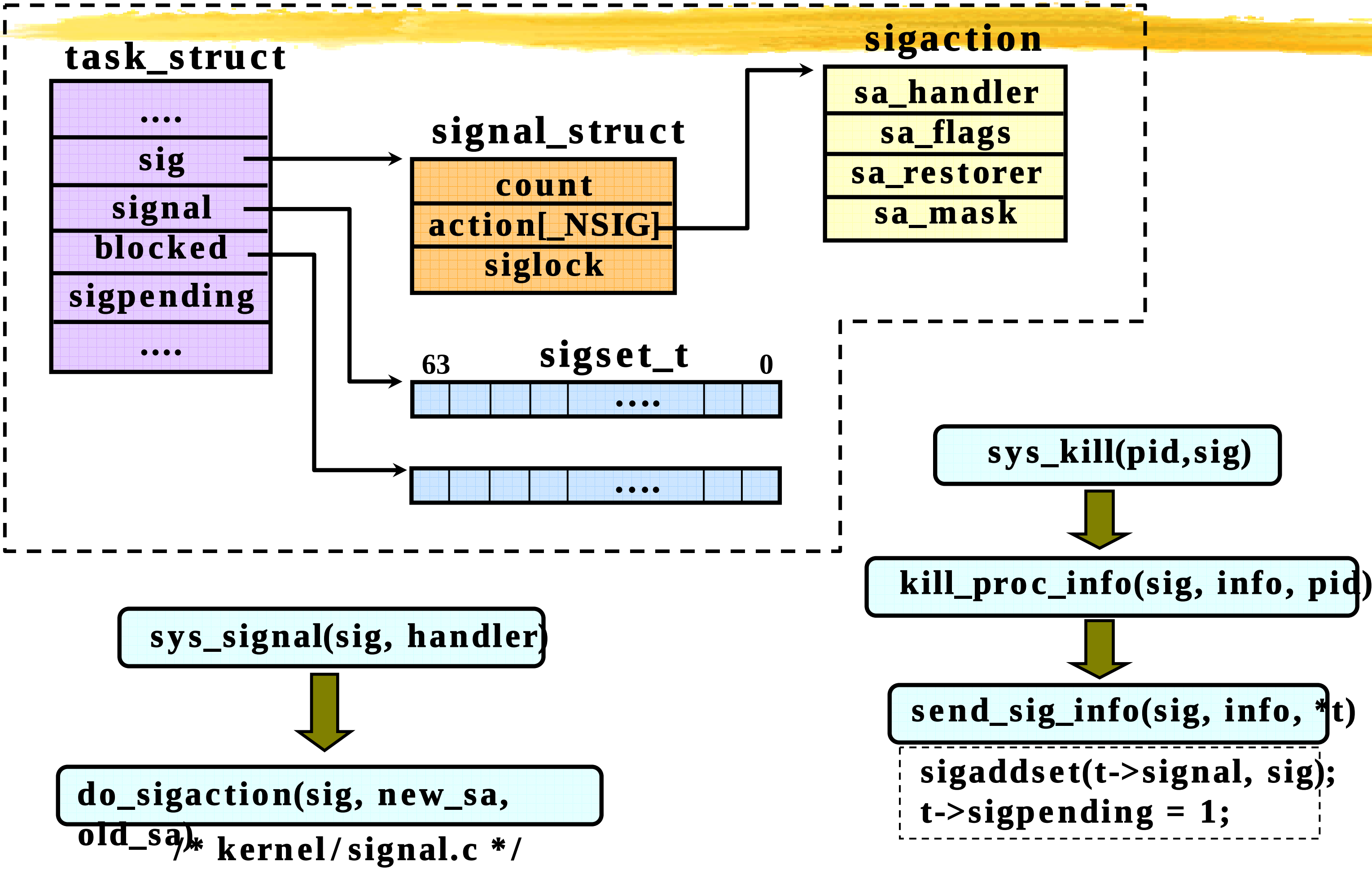
```
static inline int try_to_wake_up(struct task_struct * p, int synchronous)
{
  unsigned long flags;
  int success = 0;
  /*
   * We want the common case fall through straight, thus the goto.
   */
  spin_lock_irqsave(&runqueue_lock, flags);
  p->state = TASK_RUNNING;
  if (task_on_runqueue(p))
                goto out;
  add_to_runqueue(p);
  if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
                reschedule_idle(p);
  success = 1;
out:
  spin_unlock_irqrestore(&runqueue_lock, flags);
  return success;
}
inline int wake_up_process(struct task_struct * p)
{
                return try_to_wake_up(p, 0);
}
```

Virtual File System & Files

*struct fs_struct *fs;   /* checking information used when accessing files */*
*struct files_struct *files /* pointer to the file descriptors opened */*



Signal handling

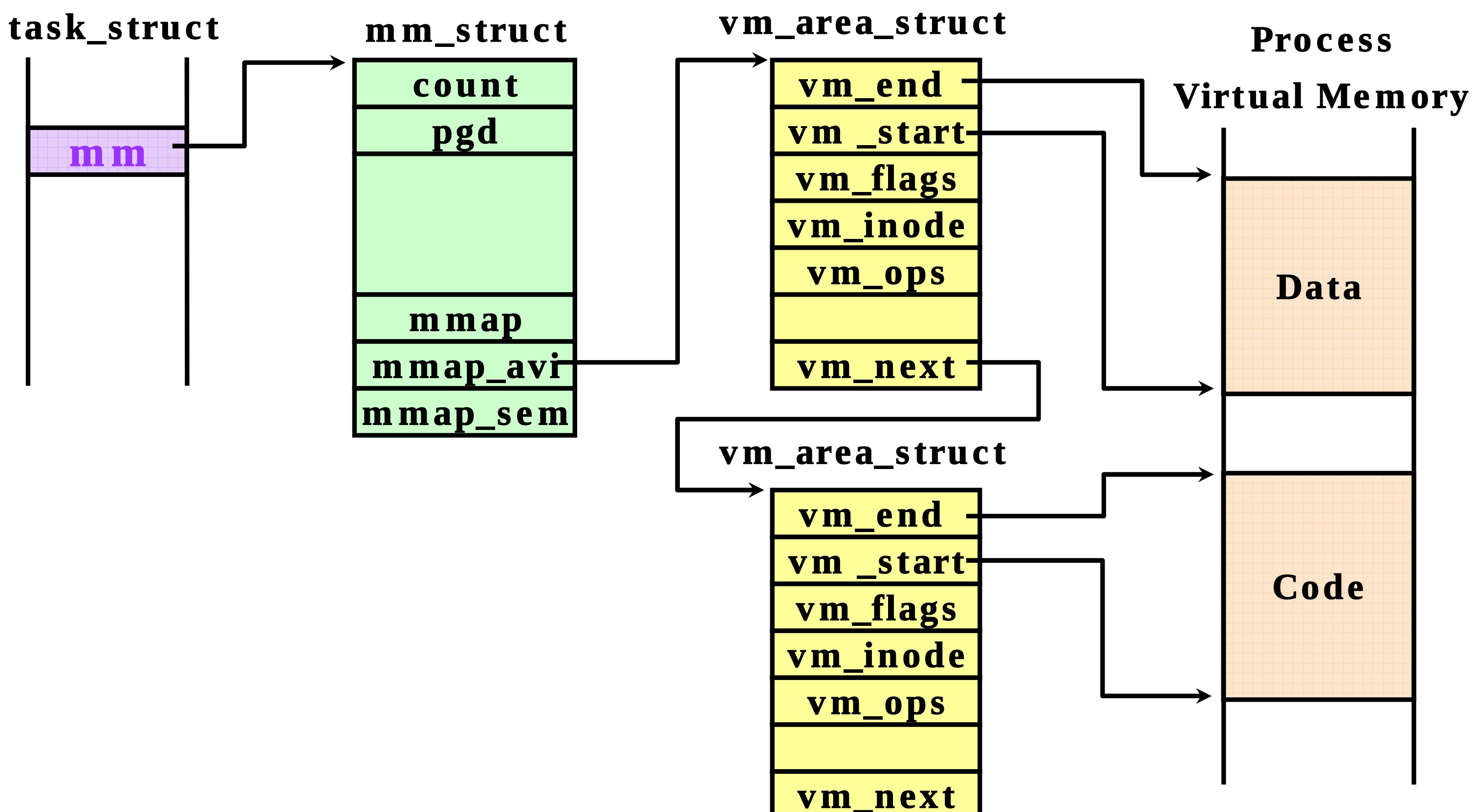


Virtual Memory

:

struct mm_struct mm   / control information used for memory management */*

:



Thread structure (CPU abstract)

task_struct

...
tss
...

include/asm-i386/processor.h

```
unsigned long  esp0;
unsigned short ss0;
unsigned long  esp1;
unsigned short ss1;
unsigned long  esp2;
unsigned short ss2;
unsigned long  cr3;
unsigned long  eip, eflags;
unsigned long  eax, ecx, edx, ebx;
unsigned long  esp;
unsigned long  ebp, esi, edi;
unsigned short  es, cs, ss, ds, fs, gs;
unsigned short ldt;
……..
```

子进程的执行

- 要让若干新进程按照需要处理不同的事情，就必须通过系统调用exec（这实际上不止是一个名为**exec**的函数；而是**exec**通常用作一个引用一系列函数的通用术语，所有这些函数基本上都处理相同的事情，但是使用的参数稍微有些不同。）
- do_execve( ) 是实现所有exec家族函数的底层内核函数。按照该函数的流程分析，它的执行过程如下：
  - 打开可执行文件,获取该文件的 file结构。
  - 获取参数区长度,调用memset()将存放参数的页面清零。memset()属于存储管理部分的函数。
  - 对linux_binprm结构的其它项作初始化。
  - linux_binprm结构用来读取并存储运行可执行文件的必要信息。

父进程的等待

- 有两种方式可以让父进程变为睡眠态等待子进程
  - 第一种情况是：在执行do_fork时，若函数的参数中的CLONE_VFORK位置一，即父子进程共享用户空间。这时，必须先运行子进程。在do_fork中系统会对父进程执行一个down( )操作，使父进程进入临界区并因为得不到临界资源而转入睡眠，从而达到等待子进程的目的。
  - 第二种情况是在应用程序中直接利用调用系统调用wait4( )来达到让父进程转入睡眠而等待子进程的目的。

子进程的消亡

- 两种消亡方式：
  - 主动方式，自行消亡
    - gcc在编连程序的时，会自动加入exit系统调用。这样，任何一个用户进程都不可避免的在运行结束时通过exit杀死自己。
  - 被动方式，被强行消灭
    - 用户可以通过给进程发送信号量9 SIGKILL强行杀掉进程。
- sys_exit( )是系统调用exit对应的内核服务程序。它的函数体内仅有一条语句，即调用do_exit( )完成具体操作。
- 自行消亡的子进程在do_exit ( )中调用exit_notify( )告知其父子进程它要消亡的消息，并把它的子进程托付给其它进程。

进程控制的全过程

- fork( )从系统调用入口ENTRY(system_call)执行系统调用总控部分的代码进入内核，然后调用sys_fork( )，进而调用do_fork( )创建一个子进程。
- 子进程创建完成后，转入调度。无论是父进程获得CPU还是子进程获得CPU，都会回到系统调用总控部分的代码的ret_from_sys_call，从这里进程返回到用户态继续执行用户程序。
  - 若从子进程返回，则返回值为0，继续执行系统调用execve( )按前述方式进入内核执行sys_execve( )，进而调用do_execve( )装入可执行文件并执行之。
  - 若从父进程返回，则返回值为子进程pid，继续执行系统调用wait( ) 按前述方式进入内核执行sys_wait( )转入睡眠，等待子进程消亡时唤醒。
- 子进程完成任务后通过系统调用exit( )进入内核执行do_exit( )自我消亡，并唤醒等待它的父进程。
- 父进程唤醒后，在sys_wait( )中清除子进程的剩余资源，把子进程从系统中完全清除。

谢谢！

子进程的消亡

/kernel/exit.c

```
asmlinkage long sys_exit(int error_code)
{
        do_exit((error_code&0xff)<<8);
}
NORET_TYPE void do_exit(long code)
{
        struct task_struct *tsk = current;

        if (in_interrupt())
                panic("Aiee, killing interrupt handler!");
        if (!tsk->pid)
                panic("Attempted to kill the idle task!");
        if (tsk->pid == 1)
                panic("Attempted to kill init!");
        tsk->flags |= PF_EXITING;
        del_timer_sync(&tsk->real_timer);

fake_volatile:
#ifdef CONFIG_BSD_PROCESS_ACC
        acct_process(code);
#endif
        __exit_mm(tsk);
```

Frees its allocated memory

/kernel/exit.c

```
static inline void __exit_mm(struct task_struct * tsk)
{
  struct mm_struct * mm = tsk->mm;
  mm_release();
  if (mm) {
    atomic_inc(&mm->mm_count);
    BUG_ON(mm != tsk->active_mm);
    /* more a memory barrier than a real lock */
    task_lock(tsk);
    tsk->mm = NULL;
    task_unlock(tsk);
    enter_lazy_tlb(mm, current, smp_processor_id());
    mmput(mm);
  }
}
```

子进程的消亡

/kernel/exit.c

```
        lock_kernel();
        sem_exit();
        __exit_files(tsk);
        __exit_fs(tsk);
        exit_namespace(tsk);
        exit_sighand(tsk);
        exit_thread();

        if (current->leader)
                disassociate_ctty(1);

        put_exec_domain(tsk->exec_domain);
        if (tsk->binfmt && tsk->binfmt->module)
                __MOD_DEC_USE_COUNT(tsk->binfmt->modu

        tsk->exit_code = code;
        exit_notify();
        schedule();
        BUG();
        goto fake_volatile;
}
```

Frees its semaphores and other System V IPC structures, source code locate at /ipc/sem.c

Release its allocated files
/kernel/exit.c

```
static inline void __exit_files(struct task_struct *tsk)
{
  struct files_struct * files = tsk->files;
  if (files) {
    task_lock(tsk);
    tsk->files = NULL;
    task_unlock(tsk);
    put_files_struct(files);
  }
}
```

子进程的消

/kernel/exit.c

```
        lock_kernel();
        sem_exit();
        __exit_files(tsk);
        __exit_fs(tsk);
        exit_namespace(tsk);
        exit_sighand(tsk);
        exit_thread();

        if (current->leader)
                disassociate_ctty(1);

        put_exec_domain(tsk->exec_domain);
        if (tsk->binfmt && tsk->binfmt->module)
                __MOD_DEC_USE_COUNT(tsk->binf

        tsk->exit_code = code;
        exit_notify();
        schedule();
        BUG();
        goto fake_volatile;
}
```

Release its file system data
/kernel/exit.c

```
static inline void __exit_fs(struct task_struct *tsk)
{
  struct fs_struct * fs = tsk->fs;
  if (fs) {
    task_lock(tsk);
    tsk->fs = NULL;
    task_unlock(tsk);
    __put_fs_struct(fs);
  }
}
```

Release its signal handler table
/kernel/signal.c

```
void exit_sighand(struct task_struct *tsk)
{
  struct signal_struct * sig = tsk->sig;
  spin_lock_irq(&tsk->sigmask_lock);
  if (sig) {
    tsk->sig = NULL;
    if (atomic_dec_and_test(&sig->count))
      kmem_cache_free(sigact_cachep, sig);
  }
  tsk->sigpending = 0;
  flush_sigqueue(&tsk->pending);
  spin_unlock_irq(&tsk->sigmask_lock);
}
```

子进程的消亡

/kernel/exit.c

```
        lock_kernel();
        sem_exit();
        __exit_files(tsk);
        __exit_fs(tsk);
        exit_namespace(tsk);
        exit_sighand(tsk);
        exit_thread();

        if (current->leader)
                disassociate_ctty(1);

        put_exec_domai
        if (tsk->binfmt && tsk->binfmt->          e)
                __MOD_DEC_USE_C     NT(tsk->binfmt->module);

        tsk->exit_code = code;
        exit_notify();
        schedule();
        BUG();
        goto fake_volatile;
}
```

The exiting task's exit code is remembered for future use by its parent

Calls the liberally commented function exit_notify, which puts the process in the TASK_ZOMBIE state and then alerts the exiting task's parent and the members of its process group to their compatriot's demise.

子进程的消亡

/kernel/exit.c

```
	lock_kernel();
	sem_exit();
	__exit_files(tsk);
	__exit_fs(tsk);
	exit_namespace(tsk);
	exit_sighand(tsk);
	exit_thread();

	if (current->leader)
		disassociate_ctty(1);

	put_exec_domain(tsk->exec_domain);
	if (tsk->binfmt && tsk->binfmt->module)
		__MOD_DEC_USE_COUNT(tsk->binfmt->module);

	tsk->exit_code = code;
	exit_notify();
	schedule();
	BUG();
	goto fake_volatile;
}
```

Calls schedule to give away the CPU. This call to schedule never returns.