

Linux操作系统分析

Chapter 9 进程管理

陈香兰 (xlanchen@ustc.edu.cn)

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

December 23, 2014

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

4 进程的创建和删除

5 进程调度

6 小结和作业

进程和线程

- 多道程序，对操作系统的需求⇒进程
- 进一步提高并发度，对操作系统的需求⇒线程

- 进程是执行程序的一个实例

- ▶ 几个进程可以并发的执行一个程序
- ▶ 一个进程可以顺序的执行几个程序

- 进程和程序的区别？线程和进程的区别？

- Linux 2.4内核以及之前的版本都不支持线程

Linux 2.6内核中有thread，但仍不是线程

- ▶ Linux中的线程主要在用户态实现，不是本课程的内容
- ▶ 但Linux内核对用户态线程有一定的辅助支持

唯一的标识一个进程

❶ 使用进程描述符地址

- ▶ 进程和进程描述符之间有非常严格的一一对应关系，使得用32位进程描述符地址标识进程非常方便

❷ 使用PID (Process ID, PID)

- ▶ 每个进程的PID都存放在进程描述符的pid域中
- ▶ 常用接口getpid()

在/proc文件系统中列出所有的进程

```
ls /proc -U
```

Outline

1 进程描述符

- Linux的进程描述符：task_struct

- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Linux的进程描述符：task_struct

- 为了管理进程，内核必须对每个进程进行清晰的描述。
- 进程描述符提供了内核所需了解的进程信息。

struct task_struct (参见源文件：include/linux/sched.h)

- ▶ 数据结构很庞大

- ★ 基本信息
- ★ 管理信息
- ★ 控制信息

- ▶ 示意图，参见ULK

- 进程描述符的分配/回收，参见kernel/fork.c

```
#ifndef __HAVE_ARCH_TASK_STRUCT_ALLOCATOR
# define alloc_task_struct() kmem_cache_alloc(task_struct_cachep, GFP_KERNEL)
# define free_task_struct(tsk) kmem_cache_free(task_struct_cachep, (tsk))
static struct kmem_cache *task_struct_cachep;
#endif
```

Linux进程的状态：task_struct::state

```
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
```

include/linux/sched.h

```
/* * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */
```

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define __TASK_STOPPED        4
#define __TASK_TRACED         8
/* in tsk->exit_state */
#define EXIT_ZOMBIE           16
#define EXIT_DEAD             32
/* in tsk->state again */
#define TASK_DEAD             64
#define TASK_WAKEKILL        128
```

state是按bit定义的
除TASK_RUNNING，其他都是状态位掩码
当不处于其他任何状态时，即TASK_RUNNING
Linux还定义了一些宏
(1)一些组合状态
(2)关于状态的判断
(3)关于状态的设置

Linux进程的pid及其分配

- task_struct中：

```
pid_t pid;  
pid_t tgid;
```

- 类型定义

```
include/linux/types.h:      typedef __kernel_pid_t pid_t;  
include/asm-arm/posix_types.h:typedef int __kernel_pid_t;
```

- pid的取值范围[0,PID_MAX_DEFAULT)

```
include/linux/threads.h:  
#define PID_MAX_DEFAULT (CONFIG_BASE_SMALL ? 0x1000 : 0x8000)
```

- 进程创建过程中，使用**alloc_pid**分配进程的pid
 - do_fork→copy_process→alloc_pid
- pid位图和pid名字空间，参见include/linux/pid_namespace.h

用户如何获得一个进程的pid

- **getpid**, **getppid** - get process identification
(注意：返回tgid)

kernel/timer.c

```
/**
 * sys_getpid - return the thread group id of the current process
 *
 * Note, despite the name, this returns the tgid not the pid. The tgid and
 * the pid are identical unless CLONE_THREAD was specified on clone() in
 * which case the tgid is the same in all threads of the same group.
 *
 * This is SMP safe as current->tgid does not change.
 */
asmlinkage long sys_getpid(void) {
    return task_tgid_vnr(current);
}

...
asmlinkage long sys_getppid(void) {
    ...
    pid = task_tgid_vnr(current->real_parent);
    ...
}
```

逻辑CPU的当前进程：current宏

- Current宏可以看成当前进程的进程描述符指针，在内核中直接使用

```
include/asm-arm/current.h
```

```
static inline struct task_struct *get_current(void) __attribute__((const));

static inline struct task_struct *get_current(void)
{
    return current_thread_info()->task;
}

#define current (get_current())
```

- ▶ 举例：比如current->pid返回在CPU上正在执行的进程的PID
- ▶ 观察内核中对current的使用情况
- ▶ 问题：current所代表的进程什么时候被切换？

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

进程的栈

- 一个普通的用户进程有2个栈

- ▶ 用户态的栈

- ★ 用户态代码的运行使用用户态的栈
 - ★ 用户态的栈在用户地址空间中

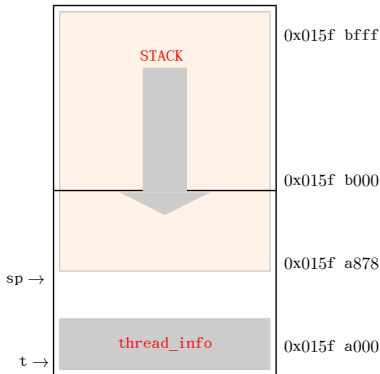
- ▶ 内核态的栈

- ★ 内核代码（**内核控制路径**）的运行使用内核态的栈
 - ★ 内核态的栈由内核进行管理和分配

进程的栈

● 内核态的栈及其大小

- ▶ Linux为每个进程分配一个8KB大小的内存区域，用于存放该进程两个不同的数据结构：
 - ① `thread_info@`
`include/asm-arm/thread_info.h`
 - ② 进程的内核堆栈
- ▶ 不同于用户态堆栈，由于内核控制路径所用的堆栈很少，因此对栈和 `thread_info` 来说，**8KB足够了**
- ▶ C语言允许用如下的一个union结构来方便的表示这样的混合体



```
include/linux/sched.h
```

```
union thread_union {  
    struct thread_info thread_info;  
    unsigned long stack[THREAD_SIZE/sizeof(long)];  
};
```

thread_info数据结构

- thread_info数据结构，参见include/asm-arm/thread_info.h
- Thread_info的分配/回收使用页面级分配器
(注：比较进程描述符的分配和回收)

参见include/asm-arm/thread_info.h

```
/* thread information allocation */
#ifdef CONFIG_DEBUG_STACK_USAGE
#define alloc_thread_info(tsk) \
    ((struct thread_info *)__get_free_pages(GFP_KERNEL | __GFP_ZERO, \
        THREAD_SIZE_ORDER))
#else
#define alloc_thread_info(tsk) \
    ((struct thread_info *)__get_free_pages(GFP_KERNEL, THREAD_SIZE_ORDER))
#endif

#define free_thread_info(info) \
    free_pages((unsigned long)info, THREAD_SIZE_ORDER);
```

thread_info数据结构

- `current_thread_info`，从内核堆栈获得当前进程的`thread_info`
 - ▶ 从刚才看到的`thread_info`和内核态堆栈之间的配对，内核可以很容易的从`sp`寄存器的值获得当前在CPU上运行的进程的`thread_info`起始地址
 - ▶ 因为这个内存区是 $8\text{KB} = 2^{13}\text{B}$ 大小，并且起始地址是8KB对齐的，内核只需让`sp`低13位为0，即可获得`thread_info`的基地址

```
include/asm-arm/thread_info.h
```

```
/*  
 * how to get the thread information struct from C  
 */  
static inline struct thread_info *current_thread_info(void) __attribute_const__;  
  
static inline struct thread_info *current_thread_info(void)  
{  
    register unsigned long sp asm (" sp" );  
    return (struct thread_info *) (sp & ~(THREAD_SIZE - 1));  
}
```


Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

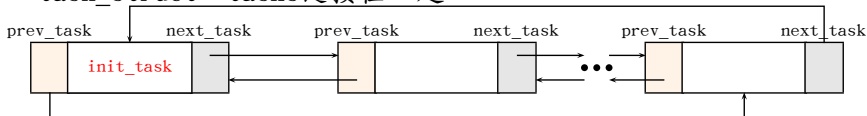
6 小结和作业

进程相关的几个链表

- 为了对给定属性的进程(比如所有在可运行状态下的进程)进行有效的搜索，内核维护了几个进程链表
 - ▶ 所有进程链表
 - ▶ TASK_RUNNING状态的进程组织：运行队列（参见进程调度）
 - ▶ pidhash表及链接表
 - ▶ 进程之间的亲属关系
 - ▶ 等待队列
- Linux内核为自己提供专门的双向链表结构和相关操作，参见include/linux/list.h和lib/list_debug.c

1、所有进程链表

- 所有进程链表以init_task为链表头，将所有进程通过task_struct::tasks链接在一起



- 新创建的进程插入链表尾部，
参见kernel/fork.c::copy_process()

```
...  
list_add_tail_rcu(&p->tasks, &init_task.tasks);  
...
```

- for_each_process宏扫描整个进程链表

```
include/linux/sched.h
```

```
#define next_task(p) list_entry(rcu_dereference((p)->tasks.next), struct task_struct,  
tasks)
```

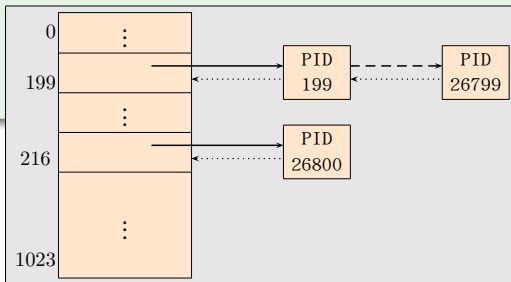
```
#define for_each_process(p) \  
for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

2、pidhash表及链接表

- 在一些情况下，内核必须能从进程的PID得出对应的进程描述符指针。例如kill系统调用
- 为了加速查找，引入了pid_hash散列表

参见kernel/pid.c

```
#define pid_hashfn(nr, ns) \  
    hash_long((unsigned long)nr + (unsigned long)ns, pidhash_shift)  
static struct hlist_head *pid_hash;  
static int pidhash_shift;  
...  
void __init pidhash_init(void){  
    ...  
}
```



3、进程之间的亲属关系

- 程序创建的进程具有父子关系，在编程时往往需要引用这样的父子关系。进程描述符中有几个域用来表示这样的关系

task_struct中表示亲属关系的几个域

```
/*
 * pointers to (original) parent process, youngest child, younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->parent->pid)
 */
struct task_struct *real_parent; /* real parent process (when being debugged) */
struct task_struct *parent; /* parent process */
...
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list */
struct task_struct *group_leader; /* threadgroup leader */
```

4、等待队列

- linux使用等待队列来组织TASK_RUNNING状态之外的进程
 - ▶ 进程的等待状态
 - ★ TASK_INTERRUPTIBLE和
 - ★ TASK_UNINTERRUPTIBLE
 - ▶ 进程状态提供的信息满足不了快速检索的需要
 - ▶ 处于等待状态的进程根据等待的原因分成很多类，每一类对应一个特定的事件。
 - ▶ 内核使用等待队列来分类组织
- 等待队列在内核中有很多用途，尤其是对中断处理、进程同步和定时用处很大
- 等待队列使得进程可以在事件的条件等待，并且当等待的条件为真时，由内核唤醒它们

4、等待队列

- 等待队列由循环链表实现

等待队列头节点，参见include/linux/wait.h

```
struct __wait_queue_head {  
    spinlock_t lock;  
    struct list_head task_list;  
};  
typedef struct __wait_queue_head wait_queue_head_t;
```

等待节点，参见include/linux/wait.h

```
typedef struct __wait_queue wait_queue_t;  
...  
struct __wait_queue {  
    unsigned int flags;  
#define WQ_FLAG_EXCLUSIVE 0x01  
    void *private;  
    wait_queue_func_t func;  
    struct list_head task_list;  
};
```

4、等待队列

- 在等待队列上内核实现了一些操作函数，参见kernel/wait.c

- ▶ 加入等待队列

- ★ add_wait_queue
- ★ add_wait_queue_exclusive

- ▶ 从等待队列删除

- ★ remove_wait_queue

```
static inline void __add_wait_queue(wait_queue_head_t *head, wait_queue_t *new) {  
    list_add(&new->task_list, &head->task_list);  
}  
  
/*  
 * Used for wake-one threads:  
 */  
static inline void __add_wait_queue_tail(wait_queue_head_t *head, wait_queue_t *new) {  
    list_add_tail(&new->task_list, &head->task_list);  
}  
  
static inline void __remove_wait_queue(wait_queue_head_t *head, wait_queue_t *old) {  
    list_del(&old->task_list);  
}
```


Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

proc文件系统简介

- 阅读Documentation/filesystems/proc.txt
- The proc file system acts as **an interface to internal data structures in the kernel**. It can be used
 - ① to obtain information about the system and
 - ② to change certain kernel parameters at runtime (sysctl).
- CHAPTER 1: COLLECTING SYSTEM INFORMATION
 - ▶ 1.1 Process-Specific Subdirectories
 - ▶ 1.2 Kernel data
 - ▶ 1.3 IDE devices in /proc/ide
 - ▶ ...
- CHAPTER 2: MODIFYING SYSTEM PARAMETERS
 - ▶ ...

Outline

- 1 进程描述符
- 2 进程的等待和唤醒
- 3 进程切换
- 4 进程的创建和删除
- 5 进程调度
- 6 小结和作业

进程等待

- 等待一个特定事件的进程能调用下面几个函数中的任一个
 - ▶ `sleep_on`
 - ▶ `sleep_on_timeout`
 - ▶ `interruptible_sleep_on`
 - ▶ `interruptible_sleep_on_timeout`
- 进程等待由需要等待的进程自己进行（调用）

进程等待

- sleep_on

sleep_on相当于

```
void sleep_on(wait_queue_head_t *wq)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(wq, &wait); /* wq points to the wait queue head */
    schedule();
    remove_wait_queue(wq, &wait);
}
```

- ▶ 实际的sleep_on系列代码，参见kernel/sched.c

进程等待

- 此外，还可能按照如下方式进行sleep

条件等待

```
DEFINE_WAIT(wait);  
prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);  
    /* wq is the head of the wait queue */  
  
...  
if (!condition)  
    schedule();  
finish_wait(&wq, &wait);
```

进程等待

- 例如事件等待wait_event→__wait_event

`__wait_event`, 参见include/linux/wait.h

```
#define __wait_event(wq, condition) \  
do { \  
    DEFINE_WAIT(__wait); \  
    \  
    for (;;) { \  
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \  
        if (condition) \  
            break; \  
        schedule(); \  
    } \  
    finish_wait(&wq, &__wait); \  
} while (0)
```

- ▶ 等待，直到事件发生（有效，或…）

进程的唤醒

- 利用wake_up或者wake_up_interruptible等一系列的宏，让等待队列中的进程进入TASK_RUNNING状态

wake_up系列定义，参见include/linux/wait.h

```
#define wake_up(x) __wake_up(x, TASK_NORMAL, 1, NULL)
#define wake_up_nr(x, nr) __wake_up(x, TASK_NORMAL, nr, NULL)
#define wake_up_all(x) __wake_up(x, TASK_NORMAL, 0, NULL)
#define wake_up_locked(x) __wake_up_locked((x), TASK_NORMAL)

#define wake_up_interruptible(x) __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_interruptible_nr(x, nr) __wake_up(x, TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_interruptible_all(x) __wake_up(x, TASK_INTERRUPTIBLE, 0, NULL)
#define wake_up_interruptible_sync(x) __wake_up_sync((x), TASK_INTERRUPTIBLE, 1)
```


进程的唤醒

● 主流程的几个关键函数

- ▶ `__wake_up`
 - `__wake_up_common`
 - 间接 → `default_wake_function`
 - `try_to_wake_up`
 - `activate_task`

Outline

- 1 进程描述符
- 2 进程的等待和唤醒
- 3 进程切换**
 - 进程上下文
 - 上下文切换
- 4 进程的创建和删除
- 5 进程调度
- 6 小结和作业

进程切换(process switching)

- 为了控制进程的执行，内核必须有能力和挂起正在CPU上执行的进程，并恢复以前挂起的某个进程的执行，这叫做**进程切换**、任务切换、上下文切换

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业





















进程上下文

- 包含了进程执行需要的所有信息
 - ▶ 用户地址空间
包括程序代码，数据，用户堆栈等
 - ▶ 控制信息
进程描述符，内核堆栈等
 - ▶ 硬件上下文

硬件上下文

- 尽管每个进程可以有自己的地址空间，但所有的进程只能**共享CPU的寄存器**。
- 因此，在恢复一个进程执行之前，内核必须确保每个寄存器装入了挂起该进程时的值。这样才能正确的恢复一个进程的执行
- **硬件上下文**：
进程恢复执行前必须装入寄存器的一组数据
 - ▶ 包括通用寄存器的值以及一些系统寄存器
- 在linux中，一个进程的上下文主要保存在thread_info，task_struct的thread_struct中，其他信息放在内核态堆栈中
 - ▶ thread_info参见include/asm-arm/thread_info.h
 - ▶ thread_struct参见include/asm-arm/processor.h

ARM寄存器集

Modes						
Privileged modes						
Exception modes						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	 R8_fiq
R9	R9	R9	R9	R9	R9	 R9_fiq
R10	R10	R10	R10	R10	R10	 R10_fiq
R11	R11	R11	R11	R11	R11	 R11_fiq
R12	R12	R12	R12	R12	R12	 R12_fiq
R13	R13	 R13_svc	 R13_abt	 R13_und	 R13_irq	 R13_fiq
R14	R14	 R14_svc	 R14_abt	 R14_und	 R14_irq	 R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		 SPSR_svc	 SPSR_abt	 SPSR_und	 SPSR_irq	 SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

thread_struct

```
union debug_insn {
    u32 arm;
    u16 thumb;
};

struct debug_entry {
    u32 address;
    union debug_insn insn;
};

struct debug_info {
    int nsaved;
    struct debug_entry bp[2];
};

struct thread_struct {
    /* fault info */
    unsigned long address;
    unsigned long trap_no;
    unsigned long error_code;
    /* debugging */
    struct debug_info debug;
};
```


thread_info

```
typedef unsigned long mm_segment_t;

...
struct thread_info {
    unsigned long flags; /* low level flags */
    int preempt_count; /* 0 => preemptable, <0 => bug */
    mm_segment_t addr_limit; /* address limit */
    struct task_struct *task; /* main task structure */
    struct exec_domain *exec_domain; /* execution domain */
    __u32 cpu; /* cpu */
    __u32 cpu_domain; /* cpu domain */
    struct cpu_context_save cpu_context; /* cpu context */
    __u32 syscall; /* syscall number */
    __u8 used_cp[16]; /* thread used copro */
    unsigned long tp_value;
    struct crunch_state crunchstate;
    union fp_state fpstate __attribute__((aligned(8)));
    union vfp_state vfpstate;
#ifdef CONFIG_ARM_THUMBEE
    unsigned long thumbee_state; /* ThumbEE Handler Base register */
#endif
    struct restart_block restart_block;
};
```

cpu_context_save

```
struct cpu_context_save {  
    __u32 r4;  
    __u32 r5;  
    __u32 r6;  
    __u32 r7;  
    __u32 r8;  
    __u32 r9;  
    __u32 s1;  
    __u32 fp;  
    __u32 sp;  
    __u32 pc;  
    __u32 extra[2]; /* Xscale 'acc' register, etc */  
};
```

堆栈中的上下文：

(1) 有些什么？

(2) 什么时候保存？

● 上下文切换的种类和时机：

- ▶ 模式切换时（中断、异常、系统调用相关）
- ▶ 进程切换时

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

进程上下文切换

- `schedule(void)`@`kernel/sched.c`函数
选择一个新的进程来运行，并调用
`context_switch()`@`kernel/sched.c`
进行上下文的切换，它进一步调用
`switch_to`@`include/asm-arm/system.h`
进行关键上下文切换
- 快速阅读`schedule()`，找到进程上下文切换的关键入口

进程上下文切换

- switch_to利用了prev和next两个参数：

注：last→prev

- ▶ prev：指向当前进程
- ▶ next：指向被调度的进程

```
/*
 * switch_to(prev, next) should switch from task 'prev' to 'next'
 * 'prev' will never be the same as 'next'. schedule() itself
 * contains the memory barrier to tell GCC not to cache 'current'.
 */
extern struct task_struct *__switch_to(struct task_struct *, struct thread_info *,
struct thread_info *);

#define switch_to(prev,next,last) \
do { \
    last = __switch_to(prev,task_thread_info(prev), task_thread_info(next)); \
} while (0)
```

进程上下文切换

arch/arm/kernel/entry-armv.S

```
/*
 * Register switch for ARMv3 and ARMv4 processors
 * r0 = previous task_struct, r1 = previous thread_info, r2 = next thread_info
 * previous and next are guaranteed not to be the same.
 */
ENTRY(__switch_to)
    add ip, r1, #TI_CPU_SAVE
    ldr r3, [r2, #TI_TP_VALUE]
    stmia ip!, {r4 - s1, fp, sp, lr} @ Store most regs on stack
#ifdef CONFIG_MMU
    ldr r6, [r2, #TI_CPU_DOMAIN]
#endif
#if __LINUX_ARM_ARCH__ >= 6
#ifdef CONFIG_CPU_32v6K
    clrex
#else
    strex r5, r4, [ip] @ Clear exclusive monitor
#endif
#endif
#endif
```

进程上下文切换

```
#if defined(CONFIG_HAS_TLS_REG)
    mcr p15, 0, r3, c13, c0, 3 @ set TLS register
#elif !defined(CONFIG_TLS_REG_EMUL)
    mov r4, #0xffff0fff
    str r3, [r4, #-15] @ TLS val at 0xffff0ff0
#endif
#ifdef CONFIG_MMU
    mcr p15, 0, r6, c3, c0, 0 @ Set domain register
#endif
    mov r5, r0
    add r4, r2, #TI_CPU_SAVE
    ldr r0, =thread_notify_head
    mov r1, #THREAD_NOTIFY_SWITCH
    bl atomic_notifier_call_chain
    mov r0, r5

    ldmia r4, {r4 - s1, fp, sp, pc} @ Load all regs saved previously
```


进程上下文切换

• 其中

```
include/asm-arm/asm-offsets.h
```

```
#define TI_FLAGS 0 /* offsetof(struct thread_info, flags) @ */  
#define TI_PREEMPT 4 /* offsetof(struct thread_info, preempt_count) @ */  
#define TI_ADDR_LIMIT 8 /* offsetof(struct thread_info, addr_limit) @ */  
#define TI_TASK 12 /* offsetof(struct thread_info, task) @ */  
#define TI_EXEC_DOMAIN 16 /* offsetof(struct thread_info, exec_domain) @ */  
#define TI_CPU 20 /* offsetof(struct thread_info, cpu) @ */  
#define TI_CPU_DOMAIN 24 /* offsetof(struct thread_info, cpu_domain) @ */  
#define TI_CPU_SAVE 28 /* offsetof(struct thread_info, cpu_context) @ */  
#define TI_USED_CP 80 /* offsetof(struct thread_info, used_cp) @ */  
#define TI_TP_VALUE 96 /* offsetof(struct thread_info, tp_value) @ */  
#define TI_FPSTATE 288 /* offsetof(struct thread_info, fpstate) @ */  
#define TI_VFPSTATE 428 /* offsetof(struct thread_info, vfpstate) @ */
```

进程上下文切换

- 阅读相关代码，并思考相关问题。
 - ▶ `thread_info`中的寄存器上下文是什么时候切换的？
 - ▶ 什么时候切换了堆栈？
 - ▶ 什么时候`next`进程真正开始执行呢？
 - ▶ ？哪里切换了进程的地址空间
- ★ 提示：从执行`switch_to`的位置往前找

Outline

- 1 进程描述符
- 2 进程的等待和唤醒
- 3 进程切换
- 4 进程的创建和删除
 - 进程的创建
 - Linux的进程创建
 - 内核线程及其创建
 - 进程树及其开始
 - 进程的终止和删除
- 5 进程调度
- 6 小结和作业

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
 - Linux的进程创建
 - 内核线程及其创建
 - 进程树及其开始
 - 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

进程的创建

- 许多进程可以并发的运行同一程序，这些进程共享内存中程序正文的单一副本，但每个进程有自己的单独的数据和堆栈区
- 一个进程可以在任何时刻可以执行新的程序，并且在它的生命周期中可以运行几个程序
- 又如，只要用户输入一条命令，shell进程就创建一个新进程
- 传统的UNIX操作系统采用统一的方式来创建进程：子进程复制父进程所拥有的资源
 - ▶ 缺点：
 - ★ 创建过程慢、效率低
 - ★ 事实上，子进程复制的很多资源是不会使用到的

进程的创建

- 现代UNIX内核通过引入三种不同的机制来解决这个问题

- ① 写时复制技术，Copy-On-Writing，COW
- ② 轻量级进程
- ③ vfork

- ④ 写时复制技术，Copy-On-Writing，COW

- ▶ 写时复制技术允许父子进程能读相同的物理页。
- ▶ 只要两者有一个进程试图写一个物理页，内核就把这个页的内容拷贝到一个新的物理页，并把这个新的物理页分配给正在写的进程

进程的创建

② 轻量级进程

允许父子进程共享许多数据结构

- ▶ 页表
- ▶ 打开文件的列表
- ▶ 信号处理

③ vfork

- ▶ 使用vfork创建的新进程能够共享父进程的内存地址空间。父进程在这个过程中被阻塞，直到子进程退出或者执行一个新的程序

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- **Linux的进程创建**
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Linux的进程创建

- Linux提供了几个系统调用来创建和终止进程，以及执行新程序
 - ▶ fork，vfork和clone系统调用创建新进程
 - ★ 其中，clone创建轻量级进程，必须指定要共享的资源
 - ▶ exec系统调用执行一个新程序
 - ▶ exit系统调用终止进程（进程也可以因收到信号而终止）

在主机系统中，查看上述几个调用的man信息

❶ fork系统调用创建一个新进程

▶ 进程的父子关系：

- ★ 调用fork的进程称为父进程
- ★ 新进程是子进程

▶ 进程几乎就是父进程的完全复制。

父子进程的上下文关系：

- ★ 它的地址空间是父进程的复制，一开始也是运行同一程序。
- ★ fork系统调用为父子进程返回不同的值

❷ 很多情况下，子进程从fork返回后，会调用exec来开始执行新的程序

▶ 这种情况下，子进程根本不需要读或者修改父进程拥有的所有资源。

- ★ 所以，fork中地址空间的复制依赖于Copy On Write技术，降低fork的开销

使用fork和exec的例子

```
If (result = fork() == 0){
    /* 子进程代码 */
    ...
    if (execve( "new_program" ,...) < 0){
        perror( "execve failed" );
        exit(1);
    } else if (result < 0){
        perror( "fork failed" );
    }
}
/* result == 子进程的pid, 父进程将会从这里继续执行 */
...
```

- 分开这两个系统调用是有好处的

- ▶ 比如服务器可以fork许多进程执行同一个程序
- ▶ 有时程序只是简单的exec，执行一个新程序
- ▶ 在fork和exec之间，子进程可以有选择的执行一系列操作以确保程序以所希望的状态运行
 - ★ 重定向输入输出
 - ★ 关闭不需要的打开文件
 - ★ 改变UID或是进程组
 - ★ 重置信号处理程序

- 若单一的系统调用试图完成所有这些功能将是笨重而低效的

- ▶ 现有的fork-exec框架灵活性更强
- ▶ 清晰，模块化强

do_fork

- 不论是fork，vfork还是clone，在内核中最终都调用了do_fork
 - ▶ 阅读sys_xxx；
 - ▶ 阅读do_fork； copy-process； ... 了解大致程序流程
 - ▶ ???子进程从哪里开始执行，它的返回值是什么？
 - ★ 阅读copy_thread(arch/arm/kernel/process.c)
 - ▶ 复制父进程的堆栈？？？
 - ★ 父进程的堆栈中有些什么？？？Fork系统调用？？？

内核栈中的上下文

- 内核栈中的上下文是发生中断、异常、系统调用时，由相关代码保存到内核栈中的
 - ▶ 存在例外：？
- 在用户进程（即父进程）调用fork()时，即发生了系统调用
 - ▶ 父进程的内核栈中保存了系统调用点的用户态上下文
 - ▶ 当父进程从内核态返回用户态时，将用此上下文来恢复到用户态执行
 - ▶ 子进程的创建发生在系统调用期间，子进程的上下文包括内核栈，将从父进程中复制

堆栈中的上下文使用pt_regs数据结构来描述

include/asm-arm/ptrace.h

```
/*  
 * This struct defines the way the registers are stored on the  
 * stack during a system call. Note that sizeof(struct pt_regs)  
 * has to be a multiple of 8.  
 */
```

```
struct pt_regs {  
    long uregs[18];  
};
```

```
#define ARM_cpsr uregs[16]  
#define ARM_pc uregs[15]  
#define ARM_1r uregs[14]  
#define ARM_sp uregs[13]  
#define ARM_ip uregs[12]  
#define ARM_fp uregs[11]  
#define ARM_r10 uregs[10]  
#define ARM_r9 uregs[9]  
#define ARM_r8 uregs[8]  
#define ARM_r7 uregs[7]  
#define ARM_r6 uregs[6]  
#define ARM_r5 uregs[5]  
#define ARM_r4 uregs[4]  
#define ARM_r3 uregs[3]  
#define ARM_r2 uregs[2]  
#define ARM_r1 uregs[1]  
#define ARM_r0 uregs[0]  
  
#define ARM_ORIG_r0 uregs[17]
```

[17]:ORIG_r0	高地址
[16]:cpsr	
[15]:pc	
[14]:1r	
[13]:sp	
[12]:ip	
[11]:fp	
r10	
r9	
r8	
r7	
r6	
r5	
r4	
r3	
r2	
r1	
r0	低地址

copy thread

```

int
copy_thread(int nr, unsigned long clone_flags, unsigned long stack_start,
            unsigned long stk_sz, struct task_struct *p, struct pt_regs *regs)
{
    struct thread_info *thread = task_thread_info(p);
    struct pt_regs *childregs = task_pt_regs(p);

    *childregs = *regs;
    childregs->ARM_r0 = 0;
    childregs->ARM_sp = stack_start;

    memset(&thread->cpu_context, 0, sizeof(struct cpu_context_save));
    thread->cpu_context.sp = (unsigned long)childregs;
    thread->cpu_context.pc = (unsigned long)ret_from_fork;

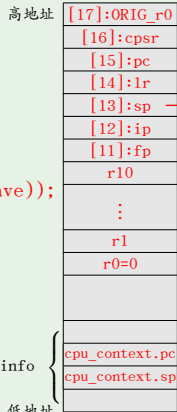
    if (clone_flags & CLONE_SETTLS)
        thread->tp_value = regs->ARM_r3;

    return 0;
}

```

子进程用
户态栈顶

子进程的内核态堆栈



```
ret from fork
```


子进程的执行

子进程什么时候就绪？

- fork后，子进程处于可运行状态，由调度器决定何时把CPU交给这个子进程
 - ▶ 进程切换后因为pc指向ret_from_fork，所以CPU立刻跳转到ret_from_fork()去执行。
 - ▶ 接着这个函数调用ret_from_sys_call()，此函数用存放在栈中的值装载所有寄存器，并强迫CPU返回用户态
 - ▶ 回忆进程的切换

子进程的执行

arch/arm/kernel/entry-common.S

```
/*
 * This is how we return from a fork.
 */
ENTRY(ret_from_fork)
    bl schedule_tail
    get_thread_info tsk
    ldr r1, [tsk, #TI_FLAGS] @ check for syscall tracing
    mov why, #1
    tst r1, #_TIF_SYSCALL_TRACE @ are we tracing syscalls?
    beq ret_slow_syscall
    mov r1, sp
    mov r0, #1 @ trace exit [IP = 1]
    bl syscall_trace
    b ret_slow_syscall
```

子进程的执行

```
/*
 * "slow" syscall return path. "why" tells us if this was a real syscall.
 */
ENTRY(ret_to_user)
ret_slow_syscall:
    disable_irq @ disable interrupts
    ldr r1, [tsk, #TI_FLAGS]
    tst r1, #_TIF_WORK_MASK
    bne work_pending
no_work_pending:
    /* perform architecture specific actions before user return */
    arch_ret_to_user r1, lr

@ slow_restore_user_regs
    ldr r1, [sp, #S_PSR] @ get calling cpsr
    ldr lr, [sp, #S_PC]! @ get pc
    msr spsr_cxsf, r1 @ save in spsr_svc
    ldmdb sp, {r0 - lr}^ @ get calling r0 - lr
    mov r0, r0
    add sp, sp, #S_FRAME_SIZE - S_PC
    movs pc, lr @ return & move spsr_svc into cpsr
```

子进程的执行

arch/arm/kernel/asm-offsets.c

```
DEFINE(S_R0, offsetof(struct pt_regs, ARM_r0));
DEFINE(S_R1, offsetof(struct pt_regs, ARM_r1));
DEFINE(S_R2, offsetof(struct pt_regs, ARM_r2));
DEFINE(S_R3, offsetof(struct pt_regs, ARM_r3));
DEFINE(S_R4, offsetof(struct pt_regs, ARM_r4));
DEFINE(S_R5, offsetof(struct pt_regs, ARM_r5));
DEFINE(S_R6, offsetof(struct pt_regs, ARM_r6));
DEFINE(S_R7, offsetof(struct pt_regs, ARM_r7));
DEFINE(S_R8, offsetof(struct pt_regs, ARM_r8));
DEFINE(S_R9, offsetof(struct pt_regs, ARM_r9));
DEFINE(S_R10, offsetof(struct pt_regs, ARM_r10));
DEFINE(S_FP, offsetof(struct pt_regs, ARM_fp));
DEFINE(S_IP, offsetof(struct pt_regs, ARM_ip));
DEFINE(S_SP, offsetof(struct pt_regs, ARM_sp));
DEFINE(S_LR, offsetof(struct pt_regs, ARM_lr));
DEFINE(S_PC, offsetof(struct pt_regs, ARM_pc));
DEFINE(S_PSR, offsetof(struct pt_regs, ARM_cpsr));
DEFINE(S_OLD_R0, offsetof(struct pt_regs, ARM_ORIG_r0));
DEFINE(S_FRAME_SIZE, sizeof(struct pt_regs));
```

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

内核线程

- 系统把一些重要的任务委托给周期性执行的进程
 - ▶ 使用top或者proc文件系统查看内核线程（主机）
 - ▶ 问题：如何查看armlinux中的内核线程？
- 内核线程与普通进程的差别
 - ▶ 每个内核线程执行一个单独指定的内核函数
 - ▶ 只运行在内核态
 - ▶ 只使用大于PAGE_OFFSET的线性地址空间
- 例如，0号进程创建1号进程init

线程和进程的比较

- Linux内核中没有线程的概念

- ▶ 没有针对所谓线程的调度策略
- ▶ 没有数据结构用来表示一个线程
- ▶ 一般线程的概念在linux中只是表现为一组共享资源的进程（每个这样的进程都有自己的进程描述符）

- 在其他系统中(比如windows)

- ▶ 线程是实实在在的一种运行抽象，提供了比进程更轻更快的调度单元
- ▶ 在linux中“线程”仅仅是表示多个进程共享资源的一种说法

创建内核线程

- `kernel_thread()` 用来创建一个内核线程，
只能由另一个内核线程来执行这个调用
 - ▶ 阅读 `kernel_thread()@arch/arm/kernel/process.c`
 - ▶ 阅读 `kernel_thread_helper()@arch/arm/kernel/process.c`

kernel_thread

观察do_fork()的stack_start和regs参数在几个调用点的区别
跟踪这两个参数的传递情况

- 1 sys_fork()
- 2 sys_clone()
- 3 sys_vfork()
- 4 kernel_thread()

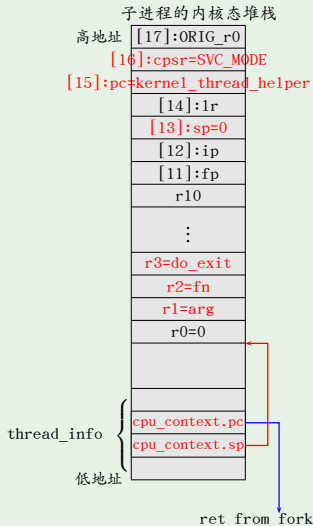
kernel_thread

```
/*
 * Create a kernel thread.
 */
pid_t kernel_thread(int (*fn)(void *), void *arg,
                    unsigned long flags)
{
    struct pt_regs regs;

    memset(&regs, 0, sizeof(regs));

    regs.ARM_r1 = (unsigned long)arg;
    regs.ARM_r2 = (unsigned long)fn;
    regs.ARM_r3 = (unsigned long)do_exit;
    regs.ARM_pc = (unsigned long)kernel_thread_helper;
    regs.ARM_cpsr = SVC_MODE;

    return do_fork(flags|CLONE_VM|CLONE_UNTRACED,
                  0, &regs, 0, NULL, NULL);
}
EXPORT_SYMBOL(kernel_thread);
```



思考：cpsr的不同情况

- ❶ 从用户态进入内核态（中断/异常/系统调用）
- ❷ 从内核态进入内核态（中断/异常/系统调用）

kernel_thread_helper

- 内核线程与常规子进程一样，从ret_from_fork开始运行，但是它不返回用户态，而是SVC_MODE
 - ▶ 按照pc值的恢复，它会执行kernel_thread_helper

```
/*
 * Shuffle the argument into the correct register before calling the
 * thread function. r1 is the thread argument, r2 is the pointer to
 * the thread function, and r3 points to the exit function.
 */
extern void kernel_thread_helper(void);
asm(    ".section .text\n"
        ".align\n"
        ".type kernel_thread_helper, #function\n"
        "kernel_thread_helper:\n"
        "    mov r0, r1\n"
        "    mov lr, r3\n"
        "    mov pc, r2\n"
        "    .size kernel_thread_helper, . - kernel_thread_helper\n"
        "    .previous" );
```

以arg为参数调用fn，当fn返回时将执行do_exit()

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

进程树及其开始

- 进程0
- 进程1
- ...

命令`ps tree`可以显示进程树

进程0 I

- 所有进程的祖先叫做进程0
- 在系统启动和初始化阶段手工创建的一个内核线程

❶ 在结构上，多数数据结构在源代码中手工建立

► `init_task@arch/arm/kernel/init_task.c`

```
/*  
 * Initial task structure.  
 *  
 * All other task structs will be allocated on slabs in fork.c  
 */  
struct task_struct init_task = INIT_TASK(init_task);
```

阅读 `include/linux/init_task.h`

进程0 II

► `init_thread_union@arch/arm/kernel/init_task.c`

```
/*
 * Initial thread structure.
 *
 * We need to make sure that this is 8192-byte aligned due to the
 * way process stacks are handled. This is done by making sure
 * the linker maps this in the .text segment right after head.S,
 * and making head.S ensure the proper alignment.
 *
 * The things we do for performance..
 */
union thread_union init_thread_union
    __attribute__((__section__(".data.init_task"))) =
    { INIT_THREAD_INFO(init_task) };
```

查看`vmlinux.lds`中关于`.data.init_task`的内容

进程0 III

- ❷ 在启动阶段，已经开始使用进程0的内核堆栈
参见arch/arm/kernel/head-common.S
 - ❸ 在启动阶段，初始化了swapper_pg_dir
 - ❹ ...
-
- 进程0最后的初始化工作创建init内核线程，此后运行cpu_idle，成为idle进程

进程1

- 1号进程，又称为init进程
- init进程是由0号进程在start_kernel@init/main.c中所调用的rest_init@init/main.c创建的，一开始是一个内核线程

```
kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
```

- init进程PID为1，当调度程序选择到init进程时，init进程开始执行kernel_init()@init/main.c函数

- kernel_init() 先进行一些必要的初始化
- 然后在init_post()中调用execve()系统调用装入可执行程序init。

从此，init内核线程变成一个普通的进程。

但init进程从不终止，因为它创建和监控操作系统外层的所有进程的活动

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

撤销进程

● 进程终止

- ▶ 进程终止的一般方式是exit()系统调用。

exit - cause normal process termination

```
#include <stdlib.h>
void exit(int status);
```

- ★ 这个系统调用可能由编程者明确的在代码中插入
- ★ 另外，在控制流到达主过程[C中的main()函数]的最后一条语句后，隐含执行exit()系统调用

回忆kernel_thread中是如何植入do_exit()调用的

- ▶ 内核可以强迫进程终止
 - ★ 当进程接收到一个不能处理或忽视的信号时
 - ★ 当在内核态产生一个不可恢复的CPU异常而内核此时正代表该进程在运行

撤销进程

- 阅读：sys_exit → do_exit

kernel/exit.c

```
asm linkage long sys_exit(int error_code)
{
    do_exit((error_code & 0xff) << 8);
}
```

- 删除进程

- ▶ 在父进程调用wait()类系统调用检查子进程是否合法终止以后，就可以删除这个进程

观察free_task_struct的调用情况

Outline

1 进程描述符

2 进程的等待和唤醒

3 进程切换

4 进程的创建和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

进程的分类

The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. Each thread has an associated scheduling policy and a static scheduling priority.

- 不同类型的进程有不同的调度需求
- 第一种分类：
 - ① I/O-bound
 - ★ 频繁的进行I/O
 - ★ 通常会花费很多时间等待I/O操作的完成
 - ② CPU-bound
 - ★ 计算密集型
 - ★ 需要大量的CPU时间进行运算

进程的分类

● 第二种分类

① 交互式进程 (interactive processes)

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

② 批处理进程 (batch processes)

- ★ 不必与用户交互，通常在后台运行
- ★ 不必很快响应
- ★ 典型的批处理程序：编译程序、科学计算

③ 实时进程 (real-time processes)

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

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Linux中的进程调度

- Linux既支持普通的分时进程，也支持实时进程
 - ▶ Linux中的调度是多种调度策略和调度算法的混合。

什么是调度策略？

是一组规则，它们决定什么时候以怎样的方式选择一个新进程运行

- ▶ Linux进程可以指定其所采用的调度策略
- Linux的调度**基于分时和优先级**
 - ▶ 随着版本的变化，分时技术在不断变化

Linux中的进程调度

- Linux的进程根据优先级排队

- ▶ 根据特定的算法计算出进程的优先级，用一个值表示
- ▶ 这个值表示把进程如何适当的分配给CPU

- Linux中进程的优先级是动态的

- ▶ 调度程序会根据进程的行为周期性的调整进程的优先级
 - ★ 较长时间未分配到CPU的进程，通常↑
 - ★ 已经在CPU上运行了较长时间的进程，通常↓

Linux的调度策略

- 必须为每个进程明确它所采用的调度策略
 - ▶ `task_struct::policy`：指明该进程采用的调度策略

Linux的调度策略

- `man sched_setscheduler`也可以看到关于调度策略的介绍

- ▶ 非实时调度策略：

- ★ `SCHED_OTHER`: the standard round-robin time-sharing policy;
- ★ `SCHED_BATCH`: for "batch" style execution of processes;
- ★ `SCHED_IDLE`: for running very low priority background jobs.

- ▶ 实时调度策略

- ★ `SCHED_FIFO`: a first-in, first-out policy;
- ★ `SCHED_RR`: a round-robin policy.

与调度相关的系统调用

- `nice` - change process priority
- `getpriority`, `setpriority` - get/set program scheduling priority (注: -20~19)
- `sched_setparam`, `sched_getparam` - set and get scheduling parameters
- `sched_setscheduler`, `sched_getscheduler` - set and get scheduling policy/parameters
- `sched_get_priority_max`, `sched_get_priority_min` - get static priority range
 - ▶ `SCHED_FIFO/SCHED_RR`: 0~99
 - ▶ `SCHED_OTHER/SCHED_BATCH`: 0
- `sched_yield` - yield the processor
- `sched_rr_get_interval` - get the `SCHED_RR` interval for the named process

改变进程的调度策略

`__setscheduler`, 参见`kernel/sched.c`

Linux的几个不同版本的调度算法

❶ Linux 2.4的调度算法

- ▶ 需要遍历可运行队列，算法 $O(n)$
- ▶ Epoch，基本时间片，动态优先级

❷ Linux 2.6.17的调度算法（2.6.23之前）

- ▶ 采用双队列（Active；expire），按照优先级组队， $O(1)$

❸ Linux 2.6.26的调度算法

- ▶ 非实时：CFS，vruntime，红黑树
- ▶ 实时：优先级队列

- 请分别阅读以上三个不同版本的调度算法的源代码

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Linux-2.6.26中调度策略

- Linux 2.6.26中的调度策略，参见include/linux/sched.h。

```
/*  
 * 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
```

- 查看linux-2.6.26中各个policy的使用情况

Linux-2.6.26中调度策略

- 0号进程的调度策略是SCHED_NORMAL
- 1号进程？

怎么查看0号进程和1号进程的调度策略？

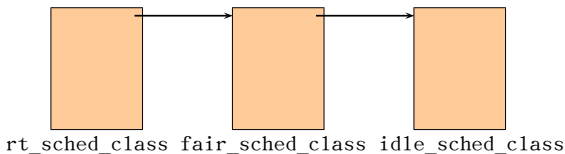
- 1 0号进程：阅读INIT_TASK
- 2 1号进程：cat /proc/1/sched

Linux-2.6.26中调度策略

- SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE的异同？
 - ① 相同：sched_class都是fair_sched_class
(一开始如此，但可以另外再调整调度类，如idle进程)
 - ② 不同：SCHED_BATCH不抢占其他进程（完全由tick驱动）
 - ③ 不同：SCHED_IDLE的优先级相关的权重最低
- SCHED_RR和SCHED_FIFO的异同
 - ① 都是实时调度策略
 - ② 不同：同优先级时，FIFO or RR

Linux-2.6.26中的sched_class调度类

- 调度类由数据结构sched_class定义
 - ▶ 阅读include/linux/sched.h中关于sched_class的定义
- 调度类实例有3种，按照调度优先级依次为：
 - ▶ rt_sched_class，参见kernel/sched_rt.c
 - ▶ fair_sched_class，参见kernel/sched_fair.c
 - ▶ idle_sched_class，参见kernel/sched_idletask.c



Linux-2.6.26中的sched_class调度类

- 哪些进程会使用idle_sched_class
 - ▶ 0号进程在进入多任务调度前，将自身的调度类调整为idle_sched_class
 - ▶ 在多个逻辑CPU的情况下，各个cpu的idle进程的调度类也是idle_sched_class

SCHED_IDLE与idle_sched_class有什么关系？

Linux-2.6.26中的就绪队列

- 每个逻辑CPU有一个单独的就绪队列，参见kernel/sched.c::rq
 - ▶ 就绪队列rq把实时任务和非实时任务分开组织：
 - ★ `rq::cfs_rq`和`rq::rt_rq`，参见kernel/sched.c
(回忆操作系统课程中的多级队列)
 - ① cfs队列采用红黑树来组织
 - ② rt队列采用优先级链表数组+位图来组织
- 就绪队列中入列的单位是调度实体
 - ▶ 实时任务和非实时任务使用不同的调度实体：
 - ★ `task_struct::rt`和`task_struct::se`
 - ▶ 数据结构`sched_entity`和`sched_rt_entity`，
参见include/linux/sched.h

阅读2.6.26的schedule函数

- `schedule()`，参见`kernel/sched.c`

`schedule()`中调度的关键:

```
...  
next = pick_next_task(rq, prev);  
...
```

- ▶ 阅读各调度类的`pick_next_task`方法

- ★ `pick_next_task_rt`，`pick_next_task_fair`，`pick_next_task_idle`

- 调度算法的关键：入列

阅读各调度类的`enqueue_task`方法

- ▶ `enqueue_task_rt`：RT根据优先级入列
- ▶ `enqueue_task_fair`：CFS根据`vruntime`的值入列

- ★ 其关键在于`vruntime`值的计算，
参见`kernel/sched_fair.c::update_curr`

- ▶ `idle`任务没有入列操作？

关于CFS的vruntime

- 理想的调度，所有的任务都是公平的。等速度的运行每个任务。
- cfs就是通过追踪这个vruntime来进行任务调度的。
它总是选 vruntime最小的进程来运行。
- 几个关键的vruntime更新之处：

1 set_task_cpu：

进程从原CPU上转移到新CPU上，需根据两个cpu上就绪队列min_vruntime的值的差距进行调整，使得进程的vruntime值能与新的调度队列中的进程具有一定的可比性

```
void set_task_cpu(struct task_struct *p, unsigned int new_cpu) {  
    ...  
    p->se.vruntime -= old_cfsrq->min_vruntime - new_cfsrq->min_vruntime;  
    ...  
}
```

关于CFS的vruntime

❷ 在__update_curr中，

```
/*
 * Update the current task's runtime statistics. Skip current tasks that
 * are not in our scheduling class.
 */
static inline void __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
unsigned long delta_exec)
{
    ...
    delta_exec_weighted = delta_exec;
    if (unlikely(curr->load.weight != NICE_0_LOAD)) {
        delta_exec_weighted = calc_delta_fair(delta_exec_weighted, &curr->load);
    }
    curr->vruntime += delta_exec_weighted;
}
```

关于CFS的vruntime

③ 在yield_task_fair中，

```
/*
 * sched_yield() support is very simple - we dequeue and enqueue.
 *
 * If compat_yield is turned on then we requeue to the end of the tree.
 */
static void yield_task_fair(struct rq *rq) {
    ...
    /*
     * Minimally necessary key value to be last in the tree:
     * Upon rescheduling, sched_class::put_prev_task() will place
     * 'current' within the tree based on its new key value.
     */
    se->vruntime = rightmost->vruntime + 1;
}
```

④ 等等

了解linux-2.6.26中进程的滴答更新

- 调度类中的方法task_tick用来在任务运行时进行滴答更新。
- 每个调度类都有自己的滴答更新方法：

④ 阅读task_tick_fair→entity_tick

```
static void entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued) {  
    /*  
     * Update run-time statistics of the 'current' .  
     */  
    update_curr(cfs_rq);  
    ...  
    if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))  
        check_preempt_tick(cfs_rq, curr);  
}
```

了解linux-2.6.26中进程的滴答更新

② 阅读task_tick_rt

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued) {
    ...
    /*
     * RR tasks need a special form of timeslice management.
     * FIFO tasks have no timeslices.
     */
    if (p->policy != SCHED_RR) return;
    if (!p->rt.time_slice) return;
    p->rt.time_slice = DEF_TIMESLICE;
    /*
     * Requeue to the end of queue if we are not the only element
     * on the queue:
     */
    if (p->rt.run_list.prev != p->rt.run_list.next) {
        requeue_task_rt(rq, p);
        set_tsk_need_resched(p);
    }
}
```

③ 阅读task_tick_idle (为空)

Outline

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

Linux2.6.26中的优先级

- 优先数范围为0~139，其中0~99为实时优先数；普通任务和批处理任务的优先数在100~139之间；优先数越大，优先级越低。

优先级的范围，参见include/linux/sched.h

```
/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL/SCHED_BATCH
 * tasks are in the range MAX_RT_PRIO..MAX_PRIO-1. Priority
 * values are inverted: lower p->prio value means higher priority.
 *
 * The MAX_USER_RT_PRIO value allows the actual maximum
 * RT priority to be separate from the value exported to
 * user-space. This allows kernel threads to set their
 * priority to a value higher than any user task. Note:
 * MAX_RT_PRIO must not be smaller than MAX_USER_RT_PRIO.
 */
#define MAX_USER_RT_PRIO 100
#define MAX_RT_PRIO MAX_USER_RT_PRIO
#define MAX_PRIO (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO (MAX_RT_PRIO + 20)
```


设置进程的优先级

- 子进程继承父进程的优先级
- 进程可以通过优先级设置相关的系统调用来调整自身或者其他进程的优先级
 - ▶ `nice` - change process priority
 - ▶ `getpriority, setpriority` - get/set program scheduling priority (注: -20~19)
 - ▶ `sched_setparam, sched_getparam` - set and get scheduling parameters
 - ▶ `sched_setscheduler, sched_getscheduler` - set and get scheduling policy/parameters
 - ▶ `sched_get_priority_max, sched_get_priority_min` - get static priority range
 - ★ `SCHED_FIFO/SCHED_RR`: 0~99
 - ★ `SCHED_OTHER/SCHED_BATCH`: 0

请使用man命令查看相关系统调用的详细说明

Linux 2.6.26 中的 nice 值和非实时任务的 static priority（静态优先级）

- Nice 值用来调整进程的优先级，Nice 值的范围在 -20~19 之间。

-

task_struct::static_prio: 非实时任务静态优先级，范围 [100, 139]
idle 的静态优先级为 MAX_PRIO (=140)

nice 值与静态优先级之间的转换，参见 kernel/sched.c

```
/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice) (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio) ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p) PRIO_TO_NICE((p)->static_prio)
```

User priority (用户优先级)

- 用户优先级的范围是[0,39]

用户优先级和静态优先级之间的转换, 参见kernel/sched.c

```
/*  
 * 'User priority' is the nice value converted to something we  
 * can work with better when scaling various scheduler parameters,  
 * it's a [ 0 ... 39 ] range.  
 */  
#define USER_PRIO(p) ((p)-MAX_RT_PRIO)  
#define TASK_USER_PRIO(p) USER_PRIO((p)->static_prio)  
#define MAX_USER_PRIO (USER_PRIO(MAX_PRIO))
```

INIT_TASK的初始优先级设置情况

参见include/linux/init_task.h

```
/*
 * INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1fffff (=2MB)
 */
#define INIT_TASK(tsk) \
{ \
    .state = 0, \
    .stack = &init_thread_info, \
    ...
    .prio = MAX_PRIO-20, \
    .static_prio = MAX_PRIO-20, \
    .normal_prio = MAX_PRIO-20, \
    .policy = SCHED_NORMAL, \
    ...
}
```

进程的实时优先级

- `task_struct::rt_priority`，范围`[0, MAX_RT_PRI0)`

调研Linux-2.6.26中对优先级翻转问题是如何解决的？

进程的动态优先级

- 进程的动态优先级prio随着自身和相关上下文（广义）的变化而变化
 - ▶ 非实时任务
 - ▶ 实时任务

④ sched_fork中对优先级的调整

参见kernel/sched.c

```
/*
 * fork()/clone()-time setup:
 */
void sched_fork(struct task_struct *p, int clone_flags)
{
    ...
    /*
     * Make sure we do not leak PI boosting priority to the child:
     */
    p->prio = current->normal_prio;
    if (!rt_prio(p->prio))
        p->sched_class = &fair_sched_class;
    ...
}
```

进程的动态优先级

❷ wake_up_new_task对优先级的调整

参见kernel/sched.c

```
/*  
 * wake_up_new_task - wake up a newly created task for the first time.  
 *  
 * This function will do some initial scheduler statistics housekeeping  
 * that must be done for every newly created context, then puts the task  
 * on the runqueue and wakes it.  
 */  
void wake_up_new_task(struct task_struct *p, unsigned long clone_flags)  
{  
    ...  
    p->prio = effective_prio(p);  
    ...  
}
```

❸ 等等

进程的当前有效优先级

计算进程的当前有效优先级, 参见kernel/sched.c

```
/*
 * Calculate the current priority, i.e. the priority
 * taken into account by the scheduler. This value might
 * be boosted by RT tasks, or might be boosted by
 * interactivity modifiers. Will be RT if the task got
 * RT-boosted. If not then it returns p->normal_prio.
 */
static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);
    /*
     * If we are RT tasks or we were boosted to RT priority,
     * keep the priority unchanged. Otherwise, update priority
     * to the normal priority:
     */
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}
```


进程的正常优先级

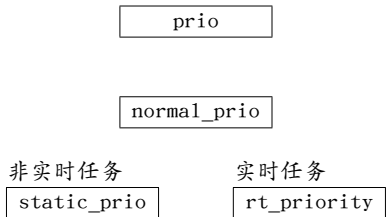
- 计算进程的正常优先级，参见kernel/sched.c

▶ 实时：实时优先级；非实时：静态优先级

```
/*
 * Calculate the expected normal priority: i.e. priority
 * without taking RT-inheritance into account. Might be
 * boosted by interactivity modifiers. Changes upon fork,
 * setprio syscalls, and whenever the interactivity
 * estimator recalculates.
 */
static inline int normal_prio(struct task_struct *p)
{
    int prio;
    if (task_has_rt_policy(p))
        prio = MAX_RT_PRIORITY - p->rt_priority;
    else
        prio = __normal_prio(p);
    return prio;
}
```

```
/*
 * __normal_prio - return the priority that is based on the static prio
 */
static inline int __normal_prio(struct task_struct *p)
{
    return p->static_prio;
}
```

几个概念之间的关系



阅读sys_nice的代码，理解nice的作用

```
/*
 * sys_nice - change the priority of the current process.
 * @increment: priority increment
 *
 * sys_setpriority is a more generic, but much slower function that
 * does similar things.
 */
asmlinkage long sys_nice(int increment)
{
    ...
    set_user_nice(current, nice);
    return 0;
}
```

```
void set_user_nice(struct task_struct *p, long nice)
{
    ...
}
```

阅读sys_setpriority的代码，参见kernel/sys.c

```
asmlinkage long sys_setpriority(int which, int who, int niceval)
{
    ...
    error = set_one_prio(p, niceval, error);
    ...
}
```

```
static int set_one_prio(struct task_struct *p, int niceval, int error)
{
    ...
    set_user_nice(p, niceval);
    ...
}
```

Outline

- 1 进程描述符
- 2 进程的等待和唤醒
- 3 进程切换
- 4 进程的创建和删除
- 5 进程调度
- 6 小结和作业

小结

1 进程描述符

- Linux的进程描述符：task_struct
- 进程的栈和thread_info数据结构
- 进程相关的几个链表
- proc文件系统简介

2 进程的等待和唤醒

3 进程切换

- 进程上下文
- 上下文切换

4 进程的创建和删除

- 进程的创建
- Linux的进程创建
- 内核线程及其创建
- 进程树及其开始
- 进程的终止和删除

5 进程调度

- 进程的分类
- Linux中的调度策略和调度算法
- Linux-2.6.26中的调度相关数据结构和代码
- Linux2.6.26中的优先级及其设置

6 小结和作业

作业

- Linux为什么要引入pidhash表？
- 在传统的UNIX系统中，创建子进程时会复制父进程的所有资源，代价比较高，现代UNIX系统中引入了哪几项技术来解决这个问题？
- Linux2.4.18中，名词解释：
 - ▶ epoch
 - ▶ 基本时间片
- Linux2.6.26中，名词解释：
 - ▶ CFS
- Linux2.6.17中，名词解释：
 - ▶ 双队列

小结

Thanks !

The end.