

进程管理及进程调度

荆琦

jingqi@pku.edu.cn

北京大学软件与微电子学院

课程回顾

一、Introduction

1. Course Introduction

2. Linux Community and OSS Introduction

3. Linux Kernel Introduction

3.1 Linux发展的五大支柱

- * UNIX
- * MINIX
- * GNU计划及自由软件
- * POSIX标准

3.2 内核发展趋势及特点

3.3 Linux发行版及软件包管理

3.4 内核源代码结构

3.5 内核编译介绍

4. Driver Introduction

课程回顾

二、Basic Knowledge of Linux Kernel Programming

1. Kernel Compiling Explanation
2. Kernel Debugging Techniques
3. Module Programming
4. AT&T Assembly Language
5. Linux Booting Process
6. Tools for Linux Kernel Source Code Reading
7. Kernel & Git

Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- Process creation
- Process Switch
- Process Termination

2. namespace

3. Process Scheduling

What is process?

“A process is just **an instance of an executing program**, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU.”

-----< *Modern operating system* >

“A process is **an instance of a computer program that is being executed**. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.”

-----< *UNIX Internals - The New Frontiers* >.

“A process is **an instance of a program running in a computer**. It is close in meaning to task , a term used in some operating systems.”

-----< *Operating systems* >.

Process in Linux

Linux has a unique implementation of threads.

- The Linux kernel does not provide any special scheduling semantics or data structures to represent threads.
- A thread is merely a process that shares certain resources with other processes.

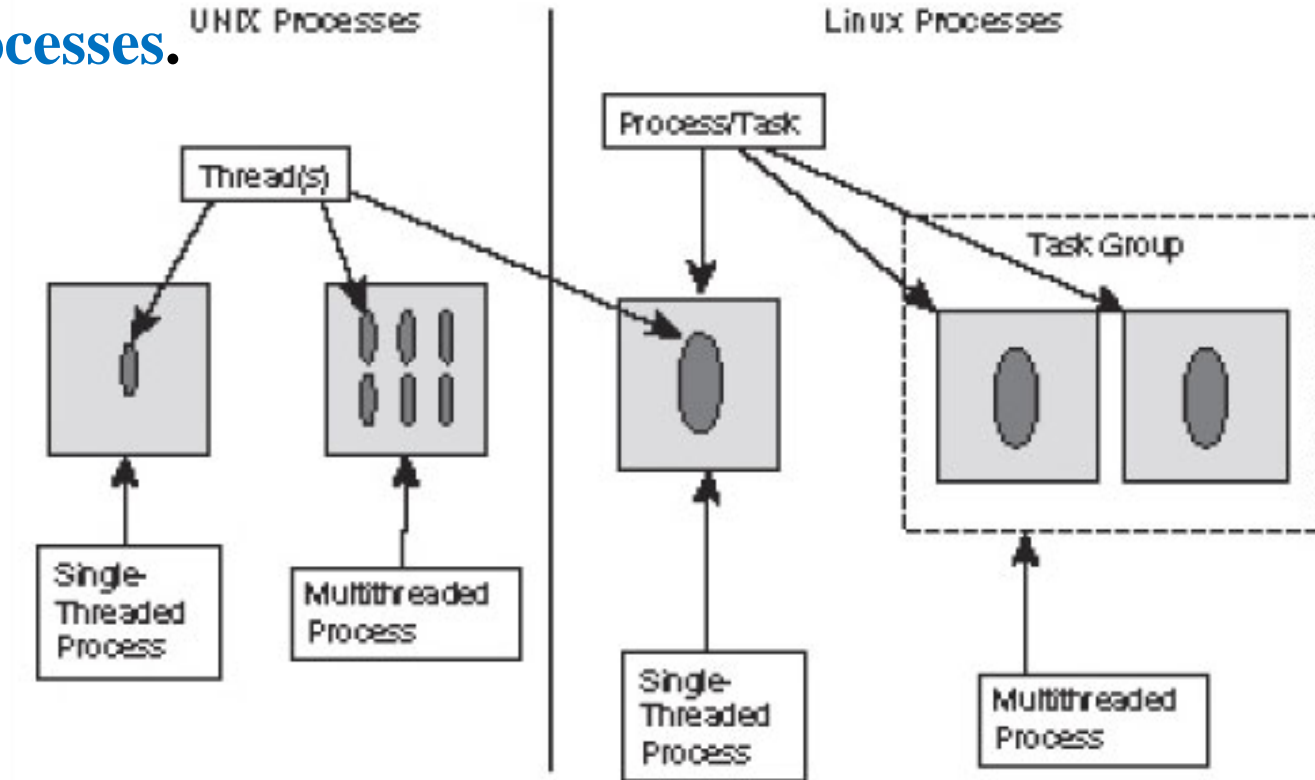


Figure 1: Comparison of UNIX and Linux processes

Cont.

进程四要素：

- (1) 有一段程序供其执行，就好像一场戏要有个剧本一样。这段程序不一定是进程所专有，可以与其它进程共用，就好像不同剧团的许多场演出可以共用一个剧本一样。
- (2) 有起码的“私有财产”，这就是进程专用的系统堆栈空间。
- (3) 有“户口”，这就是在内核中的一个 task_struct 数据结构，操作系统教科书中常称为“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，这个结构又是进程的“财产登记卡”，记录着进程所占用的各项资源。
- (4) 有独立的存储空间，意味着拥有专有的用户空间；进一步，还意味着除前述的系统空间堆栈外还有其专用的用户空间堆栈。注意，系统空间是不能独立的，任何进程都不可能直接（不通过系统调用）改变系统空间的内容（除其本身的系统空间堆栈以外）。

如果缺少第四条，则称为线程——完全没有用户空间称为“内核线程”；共享用户空间称为“用户线程”

线程

- **multithreaded process: a process is composed of several threads, each of which represents an execution flow of the process**
- 从内核角度看，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)
- 用户线程：通过**POSIX兼容的Pthreads**线程库实现
 - **LinuxThreads**
 - **NPTL (Native Posix Thread Library)**
- 内核线程：由内核创建和撤销的

POSIX标准

- (1) 查看进程列表的时候, 相关的一组线程应当被展现为一个节点;
- (2) 发送给“进程”的信号, 将被相应的一组线程共享, 并被其中的任一个“线程”处理;
- (3) 发送给某个“线程”的信号, 将只被对应的线程接收, 并且由它自己来处理;
- (4) 当“进程”被停止或继续时(对应SIGSTOP/SIGCONT信号), 对应的这一组线程状态将改变;
- (5) 当“进程”收到一个致命信号(比如由于段错误收到SIGSEGV信号), 对应的这一组线程将全部退出;
- (6)

轻量级进程

- Linux使用轻量级进程对多线程应用程序提供支持
- 轻量级进程之间可以共享一些资源（创建时指定）：
 - 地址空间；
 - 打开的文件
 - 等等

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
 - 刷新磁盘高速缓存
 - 交换出不用的页框
 - 维护网络链接等待
 - 文件系统的事务日志
 -
- 内核线程与普通进程的差别
 - 由内核创建和撤销
 - 只运行在核心态
 - 只使用大于**PAGE_OFFSET**的线性地址空间
 - 没有用户地址空间

Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- Process creation
- Process Switch
- Process Termination

2. namespace

3. Process Scheduling

Introduction

- PCB: Process Control Block
- contain all the information related to a single process
 - opened files
 - the process's address space
 - pending signals
 - the process's state
 -
- **task_struct (task_t)**
 - defined in < [include/linux/sched.h](#) >

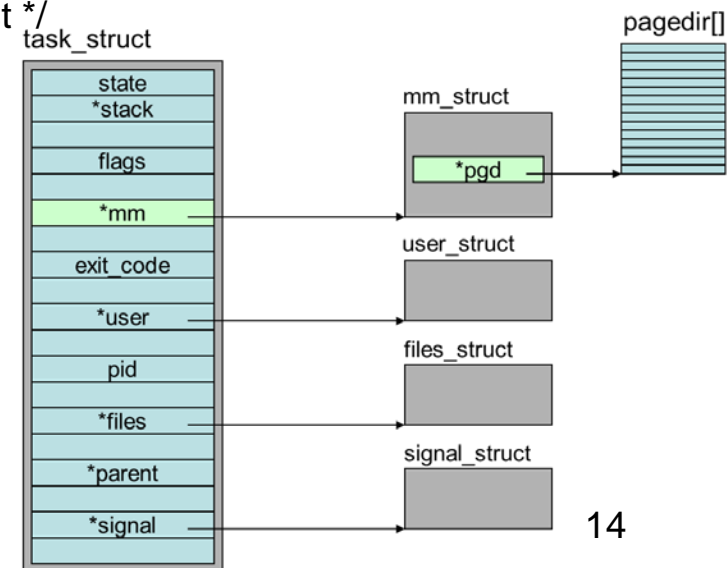
```

include/linux/sched.h
590 struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stop */
    ...
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    ...
    struct mm_struct *mm, *active_mm;
    ...
    pid_t pid;
    pid_t tgid;
    ...
    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    ...
    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 */
    ...
    /* CPU-specific state of this task */
    struct thread_struct thread;
    /* filesystem information */
    struct fs_struct *fs;
    /* open file information */
    struct files_struct *files;
    /* namespaces */
    struct nsproxy *nsproxy;
    ...
1224 };

```

during a process' lifespan, a process descriptor must keep track of:

- Process attributes
- Process relationships
- Process memory space
- File management
- Signal management
- Process credentials
- Resource limits
- Scheduling related fields

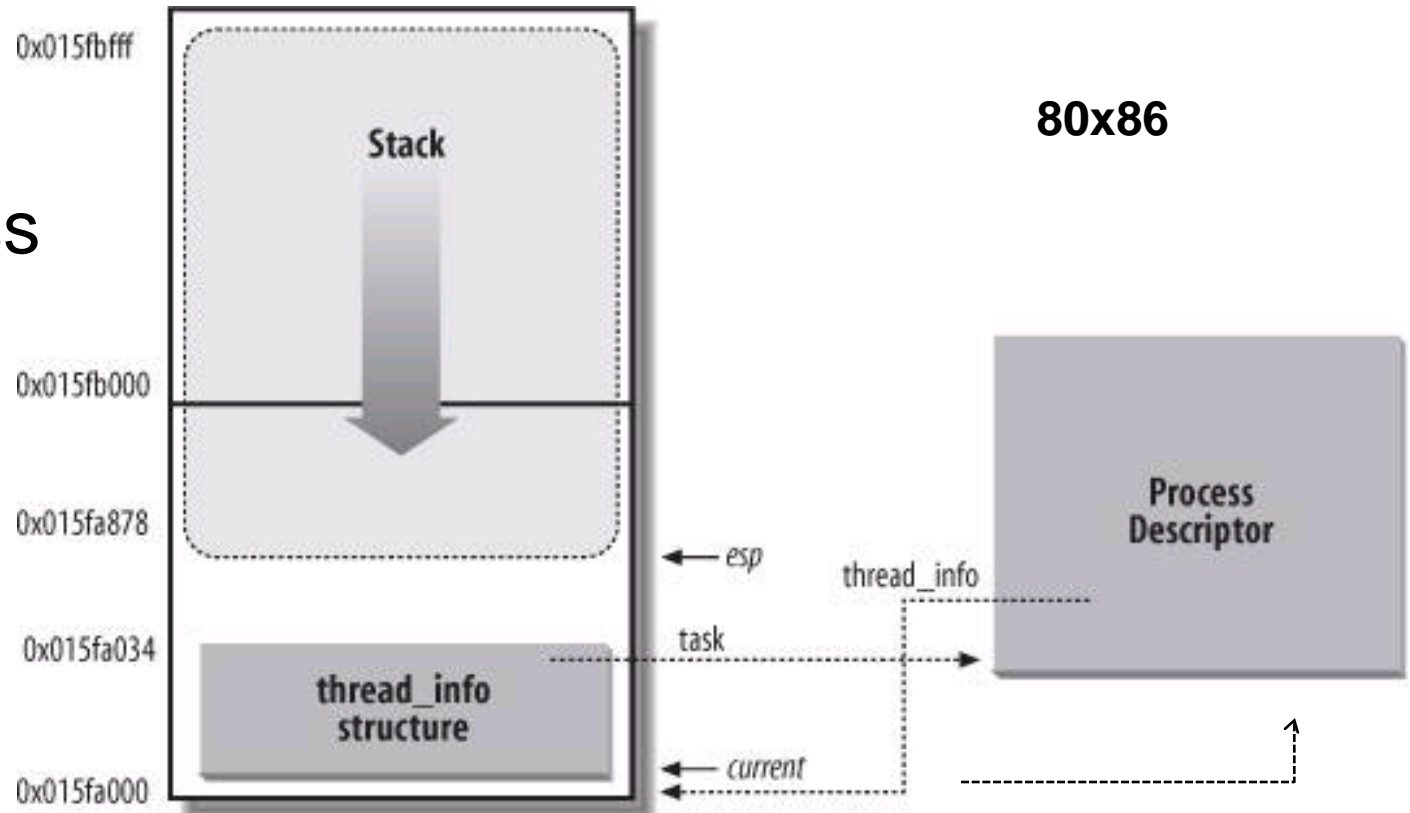


分配进程描述符

- 当创建一个新的进程时，内核为其分配**task_struct**结构
- 在**2.6**以前的内核中，每个进程的**task_struct**存放在它们内核栈的尾端
 - 内核栈：
 - 每个进程切换到内核态后使用
 - **8K or 16K (64bit)**
- 之后，在内核栈底创建一个结构**thread_info**
 - **struct thread_info**
 - **<asm/thread_info.h>**

- Linux packs two different data structures in a single **per-process** memory area:
 - thread_info structure: a small data structure linked to the process descriptor

– Kernel
Mode
process
stack



现在, **task_struct**回归内核栈

/linux/include/linux/sched.h

THREAD_SIZE:
arch/xxxx/include/asm/thread_info.h

```
1565
1566 union thread_union {
1567 #ifndef CONFIG_ARCH_TASK_STRUCT_ON_STACK
1568     struct task_struct task;
1569 #endif
1570 #ifndef CONFIG_THREAD_INFO_IN_TASK
1571     struct thread_info thread_info;
1572 #endif
1573     unsigned long stack[THREAD_SIZE/sizeof(long)];
1574 };
```

20 * On IA-64, we want to keep the task structure and kernel stack together, **so they can be**
21 * **mapped by a single TLB entry** and so they **can be addressed by the "current" pointer**
22 * without having to do pointer masking.

Identifying the Current Process

- current micro
 - get the process descriptor pointer of the process currently running on a CPU
 - different implementations on different architectures

Cont.

X86

```
#define current get_current()
```

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

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

asm-generic

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

Identifying a Process

- PID: process ID

- 局部和全局

- 全局ID: 在内核本身和初始命名空间中的唯一ID号, 在系统启动期间开始的init进程即属于初始命名空间。对每个ID类型, 都有一个给定的全局ID, 保证在整个系统中是唯一的。全局PID和TGID直接保存在task_struct中, 分别是task_struct的pid和tgid成员:

```
struct task_struct { ...  
    pid_t pid;  
    pid_t tgid;  
    ... }
```

[linux/kernel/sys.c](https://github.com/torvalds/linux/blob/master/kernel/sys.c)

```
888SYSCALL_DEFINE0(getpid)  
889{  
890     return task_tgid_vnr(current);  
891}  
892  
893/* Thread ID - the internal kernel "pid" */  
894SYSCALL_DEFINE0(gettid)  
895{  
896     return task_pid_vnr(current);  
897}
```

- 局部ID属于某个特定的命名空间, 不具备全局有效性。对每个ID类型, 它们在所属的命名空间内部有效, 但类型相同、值也相同的ID可能出现在不同的命名空间中。

```
struct pid  
{  
    atomic_t count;  
    unsigned int level;  
    /* lists of tasks that use this pid */  
    struct hlist_head tasks[PIDTYPE_MAX];  
    struct rcu_head rcu;  
    struct upid numbers[1];  
};
```

Identifying a Process

- enum pid_type

```
{  
    PIDTYPE_PID  
    PIDTYPE_PGID  
    PIDTYPE_SID  
    PIDTYPE_MAX  
};
```

进程组：其所有进程**task_struct**的**pgrp**属性值都是组长的 **PID**，可简化向成员发送信号的操作。例如通过管道连接的进程属于同一个进程组；也可使用**setpgrp**系统调用设置

会话：所有进程的**SID**一致，存储在**task_struct**的**session**中。**SID**可以使用**setsid**系统调用设置。

process states

- Possible process states (mutually exclusive)
 - TASK_RUNNING
 - The process is either executing on a CPU or waiting to be executed
 - TASK_INTERRUPTIBLE
 - The process is suspended (sleeping) until some condition becomes true or it receives a signal (back to TASK_RUNNING)
 - TASK_UNINTERRUPTIBLE
 - Like TASK_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged.
 - TASK_STOPPED
 - Process execution has been stopped (by signal SIGSTOP. The process returns to TASK_RUNNING when receiving a SIGCONT)
 - TASK_TRACED
 - Process execution has been stopped by a debugger.

Cont.

– TASK_DEAD

- EXIT_DEAD

- The final state: the process is being removed by the system

- EXIT_ZOMBIE

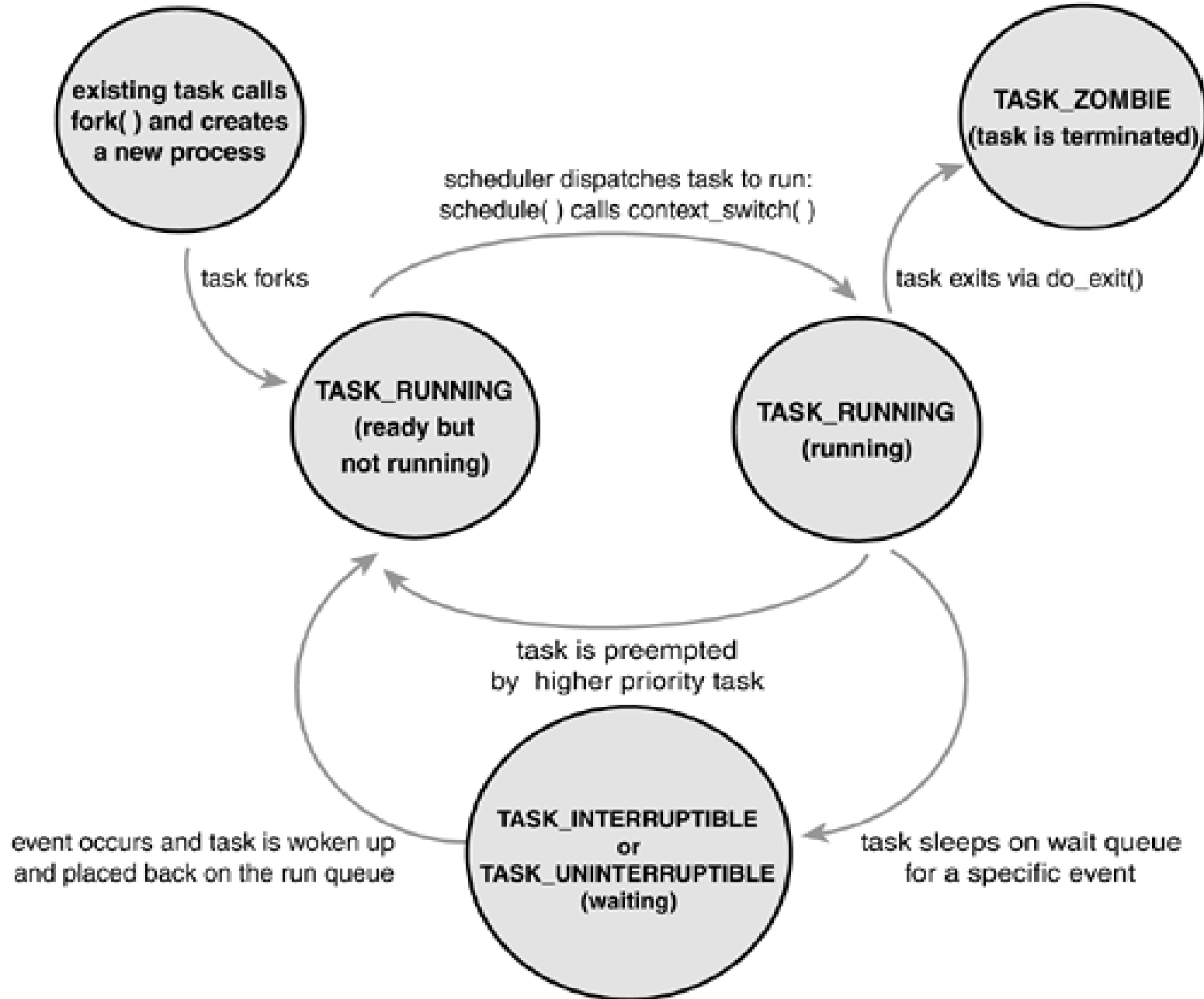
- Process execution is terminated, but the kernel cannot discard the data contained in the dead process descriptor because the parent might want to access it (the parent process has not yet issued a wait4() or waitpid() system call)

The above two states can be stored both in the **exit_state** field of the process descriptor → a process reaches one of them only when its execution is terminated.

Cont.

```
138 #define TASK_RUNNING 0
139 #define TASK_INTERRUPTIBLE 1
140 #define TASK_UNINTERRUPTIBLE 2
141 #define __TASK_STOPPED 4
142 #define __TASK_TRACED 8
143 /* in tsk->exit_state */
144 #define EXIT_ZOMBIE 16
145 #define EXIT_DEAD 32
146 /* in tsk->state again */
147 #define TASK_DEAD 64
148 #define TASK_WAKEKILL 128
149 #define TASK_WAKING 256
150 #define TASK_PARKED 512
151 #define TASK_STATE_MAX 1024
152
```

include/linux/sched.h

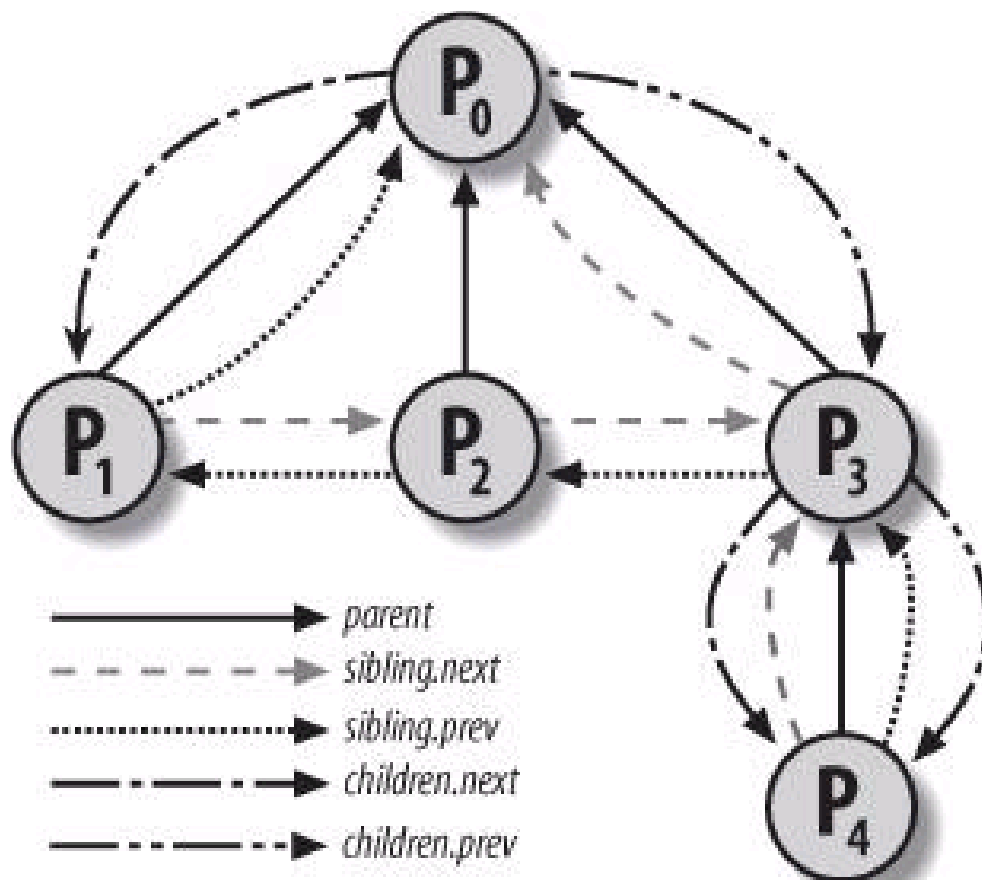


Process Family Tree

- All processes are descendents of the init process (PID=1)
 - Statically allocated as init_task

```
Struct task_struct *task;  
For (task=current; task!=init_task; task=task->parent);
```
 - Kernel starts init process in the last step of the boot process
- Every process has
 - One parent: struct task_struct *parent
 - Real_parent: process created P or process 1 (init) if the parent process no longer exists
 - Parent: current parent of P (the process that must be signaled when the child process terminates)
 - Zero or more children: struct list_head children
- Siblings: parent's direct children
 - struct list_head sibling
- The relationships stored in the process descriptor

Cont.



- 用pstree查看进程父子关系

Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- **Process creation**
- Process Switch
- Process Termination

2. namespace

3. Process Scheduling

Process Creation

- Traditional Unix systems:
 - `fork()`, creates a child process that is a copy of the current task
 - resources owned by the parent process are duplicated in the child process (difference: PID, parent PID, certain resources and statistics, such as pending signals) → process creation very slow and inefficient
 - in many cases, child process issues an immediate **execve()** to load a new executable into the address space and begins executing it (so wipe out the address space)

Cont.

- Modern Unix kernels solve the problem (“process creation very slow and inefficient”) by introducing three different mechanisms:
 - **Copy On Write technique** allows both the parent and the child to read the same physical pages. Whenever either one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process
 - **Lightweight processes** allow both the parent and the child to share many per-process kernel data structures
 - **vfork() system call** creates a process that shares the memory address space of its parent

fork(), vfork(), clone()

- fork()
 - 复制全部资源（COW），不共享
 - clone(SIGCHLD, 0)
- vfork()
 - 在调用exec 或exit 之前与父进程共享数据段
 - clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0)

```

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

```

```

asmlinkage int sys_vfork(struct pt_regs regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp
, &regs, 0, NULL, NULL);
}

```

```

sys_clone(unsigned long clone_flags, unsigned long newsp,
void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
{
    if (!newsp)
        newsp = regs->sp;
    return do_fork(clone_flags, newsp, regs, 0, parent_tid, child_tid);
}

```


do_fork()

- fork(), vfork(), clone() 在内核中最终调用do_fork() 完成主要工作
- do_fork()定义在<kernel/fork.c>
- Do_fork()参数:
 - **clone_flags**:
 - The low byte specifies **the signal number to be sent to the parent process when the child terminates**; the SIGCHLD signal is generally selected.
 - The remaining three bytes encode a group of clone flags
 - stack_start: Specifies the **User Mode stack pointer** to be assigned to the esp register of the **child process**. The invoking process (the parent) should always allocate a new stack for the child
 - regs: Pointer to the values of the general purpose registers saved into the Kernel Mode stack when switching from User Mode to Kernel Mode
 - stack_size: Unused (always set to 0)
 - parent_tidptr: Specifies the address of a User Mode variable of the parent process that will hold the PID of the new lightweight process. Meaningful only if the CLONE_PARENT_SETTID flag is set.
 - child_tidptr: Specifies the address of a User Mode variable of the new lightweight process that will hold the PID of such process. Meaningful only if the CLONE_CHILD_SETTID flag is set

Clone()参数标志

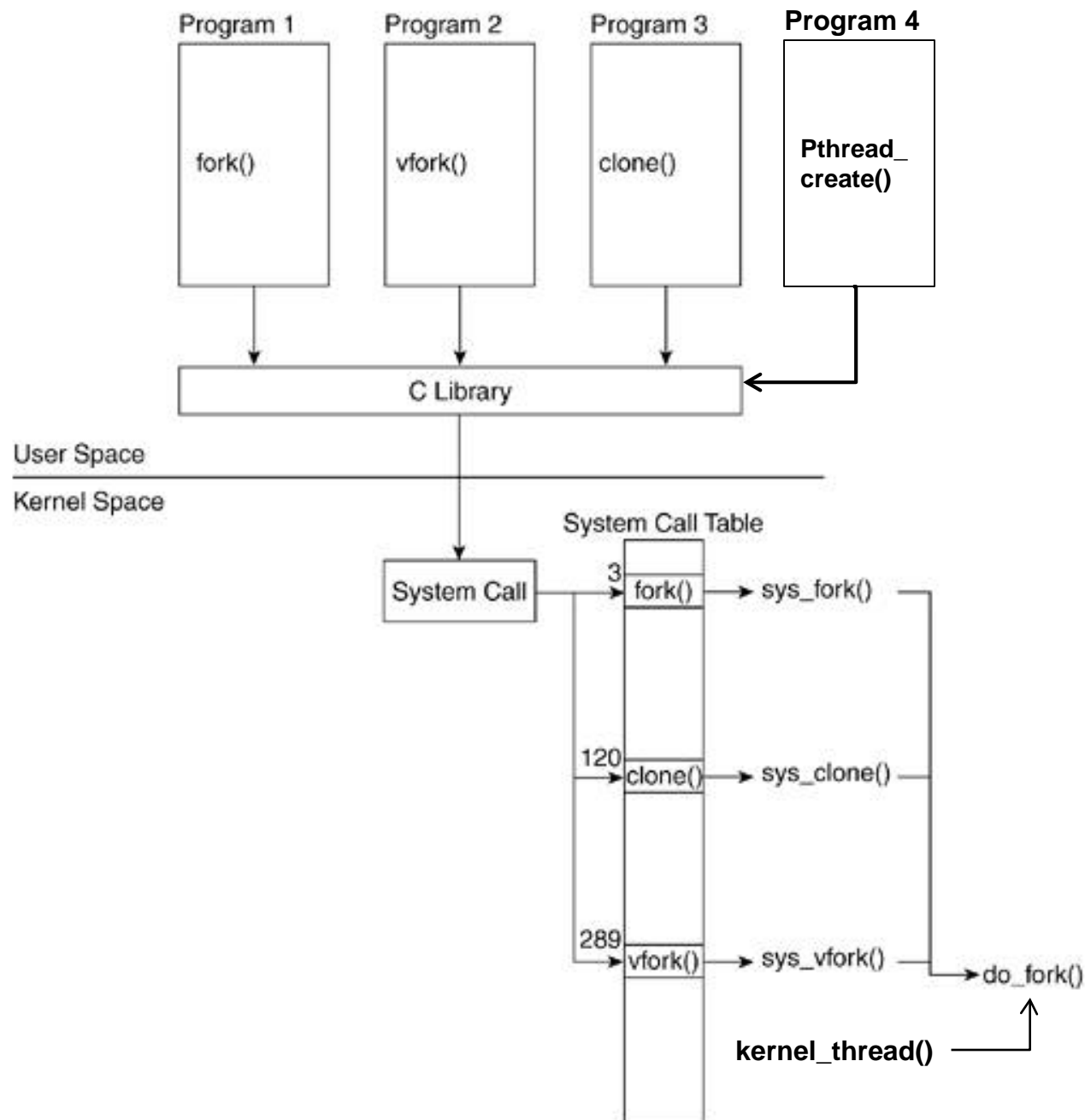
参数标志	含 义
CLONE_FILES	父子进程共享打开的文件
CLONE_FS	父子进程共享文件系统信息
CLONE_IDLETASK	将PID设置为0（只供idle进程使用）
CLONE_NEWNS	为子进程创建新的命名空间
CLONE_PARENT	指定子进程与父进程拥有同一个父进程
CLONE_PTRACE	继续调试子进程
CLONE_SETTID	将TID回写至用户空间
CLONE_SETTLS	为子进程创建新的TLS
CLONE_SIGHAND	父子进程共享信号处理函数
CLONE_SYSVSEM	父子进程共享System V SEM_UNDO语义
CLONE_THREAD	父子进程放入相同的线程组
CLONE_VFORK	调用vfork()，所以父进程准备睡眠等待子进程将其唤醒
CLONE_UNTRACED	防止跟踪进程在子进程上强制执行CLONE_PTRACE
CLONE_STOP	以TASK_STOPPED状态开始进程
CLONE_SETTLS	为子进程创建新的TLS(thread-local storage)
CLONE_CHILD_CLEARTID	清除子进程的TID
CLONE_CHILD_SETTID	设置子进程的TID
CLONE_PARENT_SETTID	设置父进程的TID
CLONE_VM	父子进程共享地址空间

```

long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
{
    struct task_struct *p;
    int trace = 0;
    long nr;
    .....
    p = copy_process(clone_flags, stack_start, regs, stack_size,
                      child_tidptr, NULL, trace);
    .....

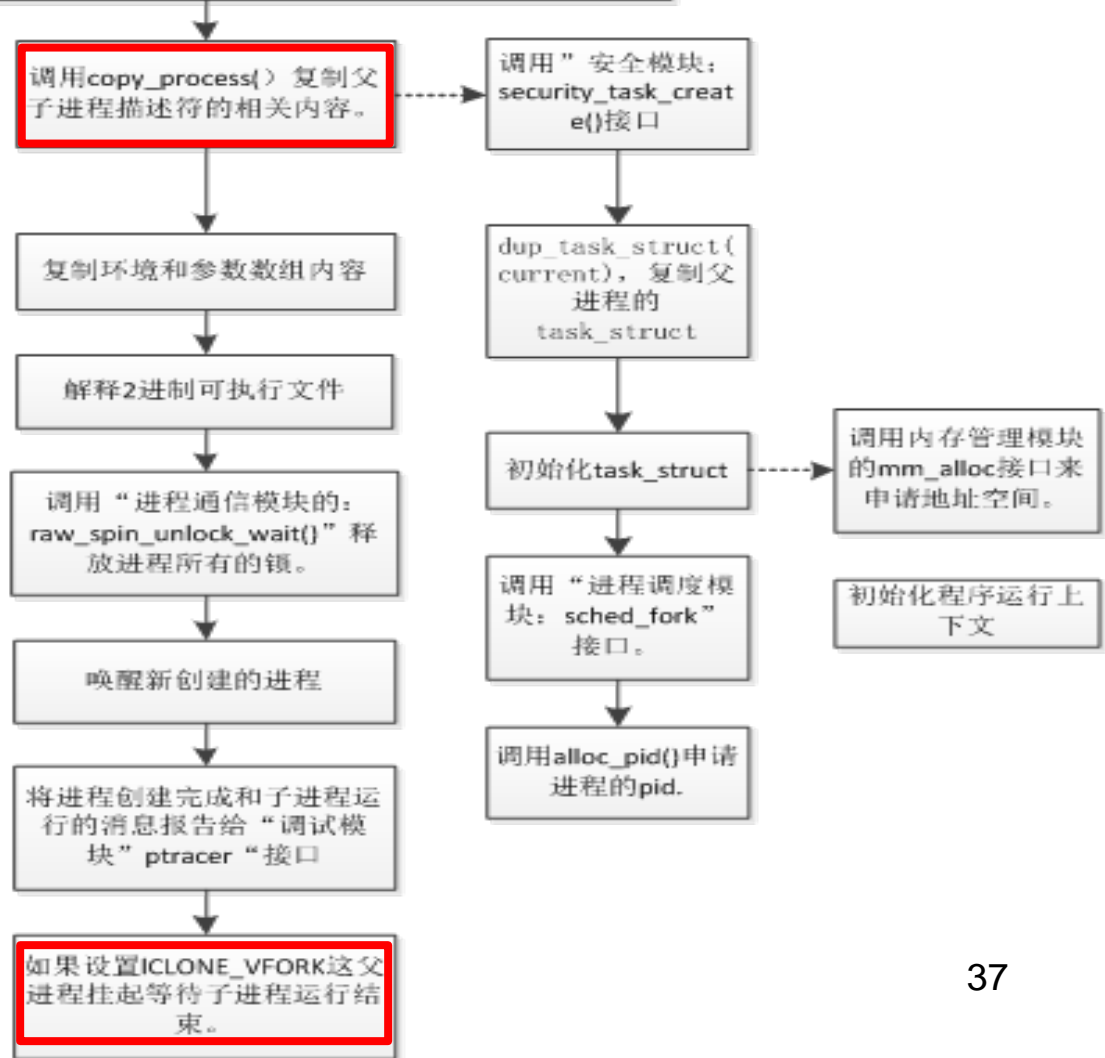
```

The `copy_process()` function sets up the process descriptor and any other kernel data structure required for a child's execution.



• 请 阅 读
do_fork ,
了解大致
流程

```
long do_fork(unsigned long clone_flags,  
             unsigned long stack_start,  
             struct pt_regs *regs,  
             unsigned long stack_size,  
             int __user *parent_tidptr,  
             int __user *child_tidptr)
```



- **ps**可以显示系统中的进程

试一下: **ps aux**

```
$ sleep 10 &
```

```
[1] 22789
```

```
$ ps -f
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
max	21341	21340	0	10:42	pts/16	00:00:00	bash
max	22789	21341	0	17:30	pts/16	00:00:00	sleep 10
max	22790	21341	0	17:30	pts/16	00:00:00	ps -f

```

int main( void )
{
    pid_t childpid;
    int status;

    childpid = fork();

    if ( -1 == childpid )
    {
        perror( "fork()" );
        exit( EXIT_FAILURE );
    }
    else if ( 0 == childpid )
    {
        puts( "In child process" );
        sleep( 3 );//让子进程睡眠3秒，看看父进程
的行为
        printf( "\tchild pid = %d\n", getpid() );
        printf( "\tchild ppid = %d\n", getppid() );
        exit( EXIT_SUCCESS );
    }
    else
    {
        waitpid( childpid, &status, 0 );
        puts( "in parent" );
        printf( "\tparent pid = %d\n", getpid() );
        printf( "\tparent ppid = %d\n",
getppid() );
        printf( "\tchild process exited with
status %d\n", status );
    }
    exit( EXIT_SUCCESS );
}

```

```

[root@localhost src]# gcc waitpid.c
[root@localhost src]# ./a.out
In child process
    child pid = 4469
    child ppid = 4468
in parent
    parent pid = 4468
    parent ppid = 4379
    child process exited with status 0
[root@localhost src]#

```

线程的创建

- 与普通的进程创建类似，调用**clone()**
 - **CLONE_THREAD**

线程组:

1. 以标志**CLONE_THREAD**来调用**clone**
2. 所有子进程都有统一的**TGID**
3. 线程组中的主进程为**group leader**，所有通过**clone**创建的线程的**task_struct**的**group_leader**都指向主进程的**task_struct**

内核线程的创建

- `kernel_thread`
 - `int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`
 - 构建`pt_regs`实例，指定其中寄存器的值
 - `Do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, ®s, 0, NULL, NULL)`
- `kthread_create`
 - `kthreadd`→`create_kthread`→`kernel_thread`
 - 创建的线程通过`wake_up_process`启动
- `kthread_run`
 - 调用`kthread_create`创建，并立刻唤醒

ps的输出中有方括号标识，“/”后是绑定的cpu号

Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- Process creation
- **Process Switch**
- Process Termination

2. namespace

3. Process Scheduling

Process Switch

- 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

进程上下文

- 包含进程执行所需的信息
 - 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
 - 每个进程可以有自己地址空间，但所有的进程只能共享CPU的寄存器。
 - 在进程恢复执行之前，内核必须确保每个寄存器装入了挂起进程时的值。
 - 包括通用寄存器的值以及一些系统寄存器
 - 通用寄存器如eax，ebx等
 - 系统寄存器如eip，esp，cr3等等

硬件上下文

- X86: TSS
- Linux: 每CPU一个TSS, 用于用户态堆栈和内核态堆栈切换
 - 一个进程的硬件上下文(体系结构相关)主要保存在**thread_struct**中
 - Task_struct中

```
/* CPU-specific state of this task */  
struct thread_struct thread;
```

thread_struct

```
447 struct thread_struct {
448     /* Cached TLS descriptors: */
449     struct desc_struct    tls_array[GDT_ENTRY_TLS_ENTRIES];
450     unsigned long         sp0;
451     unsigned long         sp;
452 #ifdef CONFIG_X86_32
453     unsigned long         sysenter_cs;
454 #else
455     unsigned long         usersp; /* Copy from PDA */
456     unsigned short        es;
457     unsigned short        ds;
458     unsigned short        fsindex;
459     unsigned short        gsindex;
460 #endif
461 #ifdef CONFIG_X86_32
462     unsigned long         ip;
463 #endif
464 #ifdef CONFIG_X86_64
465     unsigned long         fs;
466 #endif
467     .....
487     unsigned int          saved_fs;
488     unsigned int          saved_gs;
489 #endif
490     /* IO permissions: */
491     unsigned long         *io_bitmap_ptr;
492     unsigned long         iopl;
493     /* Max allowed port in the bitmap, in bytes: */
494     unsigned               io_bitmap_max;
495 };
```

arch/x86/include/asm/processor.h

上下文切换

- 在__schedule()函数中

```
if (likely(prev != next)) {  
    rq->nr_switches++;  
    rq->curr = next;  
    ++*switch_count;  
    context_switch(rq, prev, next); /* unlocks the rq */  
    cpu = smp_processor_id();  
    rq = cpu_rq(cpu);  
} else  
    raw_spin_unlock_irq(&rq->lock);
```

- 在context_switch()函数中

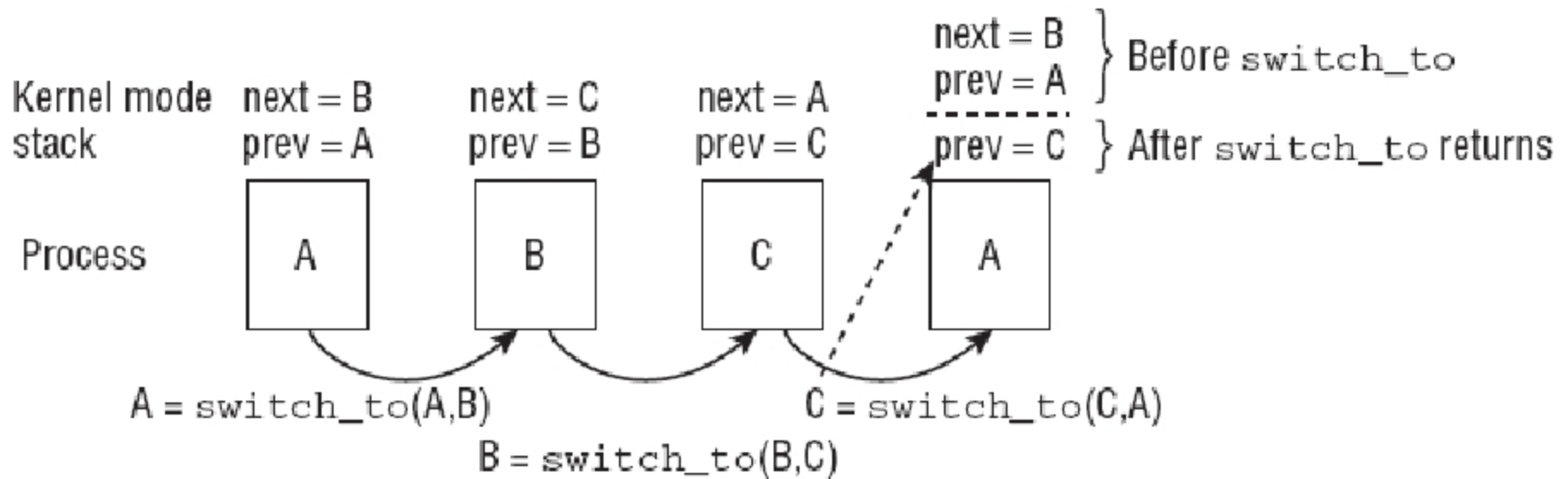
```
switch_to(prev, next, prev);
```

上下文切换

- **switch_to**宏 (in arch/x86/include/asm/switch_to.h) 执行进程切换
 - **schedule()**函数调用这个宏
 - 调度一个新的进程在**CPU**上运行
- **switch_to**利用了**prev**,**next**和**last**三个参数:
 - **prev**: 指向当前进程
 - **next**: 指向被调度的进程
 - **Last**: 指向切换回当前进程之前的那个进程

one of the most hardware-dependent routines of the kernel

switch_to (prev, next, prev)



Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- Process creation
- Process Switch
- Process Termination

2. namespace

3. Process Scheduling

Process Termination

- There are two system calls that terminate a User Mode application:
- `exit_group()`
 - terminates a full **thread group**, that is, a whole multithreaded application
 - Corresponding main kernel function is **do_group_exit()**.
 - should be invoked by the **exit()** C library function
- `exit()`
 - terminates a **single process**, regardless of any other process in the thread group
 - Corresponding main kernel function is **do_exit()**
 - invoked, for instance, by the **pthread_exit()** function of the LinuxThreads library

Cont.

- 进程的析构发生在调用**exit()**之后
- 可能**显式**调用，也可能**隐式**地在主函数的返回处。
- 当进程接受到它既不能处理也不能忽略的信号或异常时，可能**被动**地终结。
- **do_exit()** <kernel/exit.c>

```
void do_exit(long code)
{
    struct task_struct *tsk = current;
    int group_dead;

    profile_task_exit(tsk);
}
```

Cont.

- 在调用了`do_exit()`之后，尽管线程已经僵死不能再运行了，但是系统还保留了它的进程描述符。
- 系统可以在子进程终结后仍能获得它的信息。
- 进程终结所需的清理工作与进程描述符的删除工作分开。

删除进程

- 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
 - 父进程终止
- 孤儿进程
 - 父进程在子进程之前退出
 - `forget_original_parent()` → `find_new_reaper()` 寻找一个进程作为它的父亲

进程的一生

- 随着一句fork，一个新进程呱呱落地，但它这时只是老进程的一个克隆。
- 然后随着exec，新进程脱胎换骨，离家独立，开始了为人民服务的职业生涯。
- 人有生老病死，进程也一样：
 - 自然死亡，即运行到main函数的最后一个“}”，从容地离我们而去。
 - 自杀（自杀有2种方式）

这就是进程完整的一生。

 - 调用exit函数；
 - 在main函数内使用return；无论哪一种方式，它都可以留下遗书，放在返回值里保留下来。
 - 被谋杀，被其它进程通过另外一些方式结束他的生命。
- 进程死掉以后，会留下一具僵尸，wait充当了殓尸工，把僵尸推去火化，使其最终归于无形。

Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- Process creation
- Process Switch
- Process Termination

2. namespace

3. Process Scheduling

虚拟化

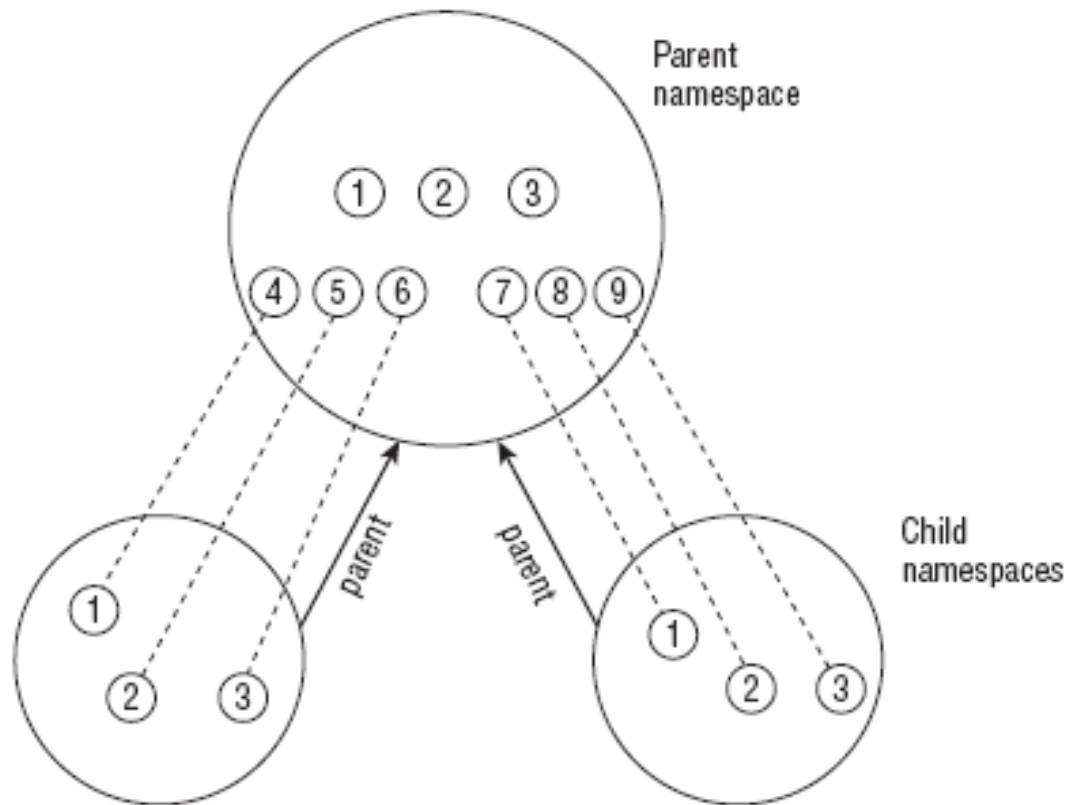
- 硬件环境
 - 成本高
- KVM or VMWare or.....
 - 独立内核/用户
 - 完整应用环境/用户
- 命名空间
 - 多命名空间/内核
 - 资源抽象
 - 隔离/共享

Linux 容器技术(Linux Containers, LXC)

- Linux 操作系统上提供的一种操作系统级虚拟化 (Operating-system-level virtualization) 技术。
- 允许在同一 Linux 内核的基础上对不同进程族的环境进行划分和隔离。
 - 这些环境包括进程 id ， 网络栈， 用户 id 和文件系统等
- 容器技术使用 Linux 内核提供的 namespace 机制隔离应用环境， 同时使用 cgroups 机制限制进程资源
- Docker 是 Docker.Inc 发布的一款开源 Linux 容器引擎， 与2013年3月首次发布。

Linux cgroups 机制

- **cgroups** 最早在2006年开始由 Google 的工程师发起，并在2008年1月首次合并到 Linux 主线内核版本 2.6.14
- 目的：为加入控制组的进程提供资源的**限制**和**审计**
 - 这里的”资源”包括进程所使用的 CPU，内存，磁盘 I/O 和网络。
- Linux 将 cgroup 实现为一个文件系统，通常位于 /sys/fs/cgroup 下
 - 目录下包含 cpu，memory，devices 等**资源**
 - 可以在相应资源下创建分组写入**资源限制**
 - 然后将**进程**加入分组来对进程资源进行控制。



- **Order**
 - **Hierarchy**
 - **Flat**
- **init**

支持

- 在编译时启用
 - 逐一指定需要支持的命名空间
 - 一般性支持总是会编译到内核中
- 默认命名空间
 - 没有显式指定的话，则每个进程关联到一个默认的命名空间
 - 作用类似于不启用

创建

- 创建新进程
 - **Flag**: 与父进程共享命名空间 or 建立新命名空间
- **unshare**系统调用
 - **分离**进程上下文中与其他进程**共享**的部分，包括命名空间

Cont.

<sched.h>

#define CLONE_NEWUTS 0x04000000

#define CLONE_NEWIPC 0x08000000

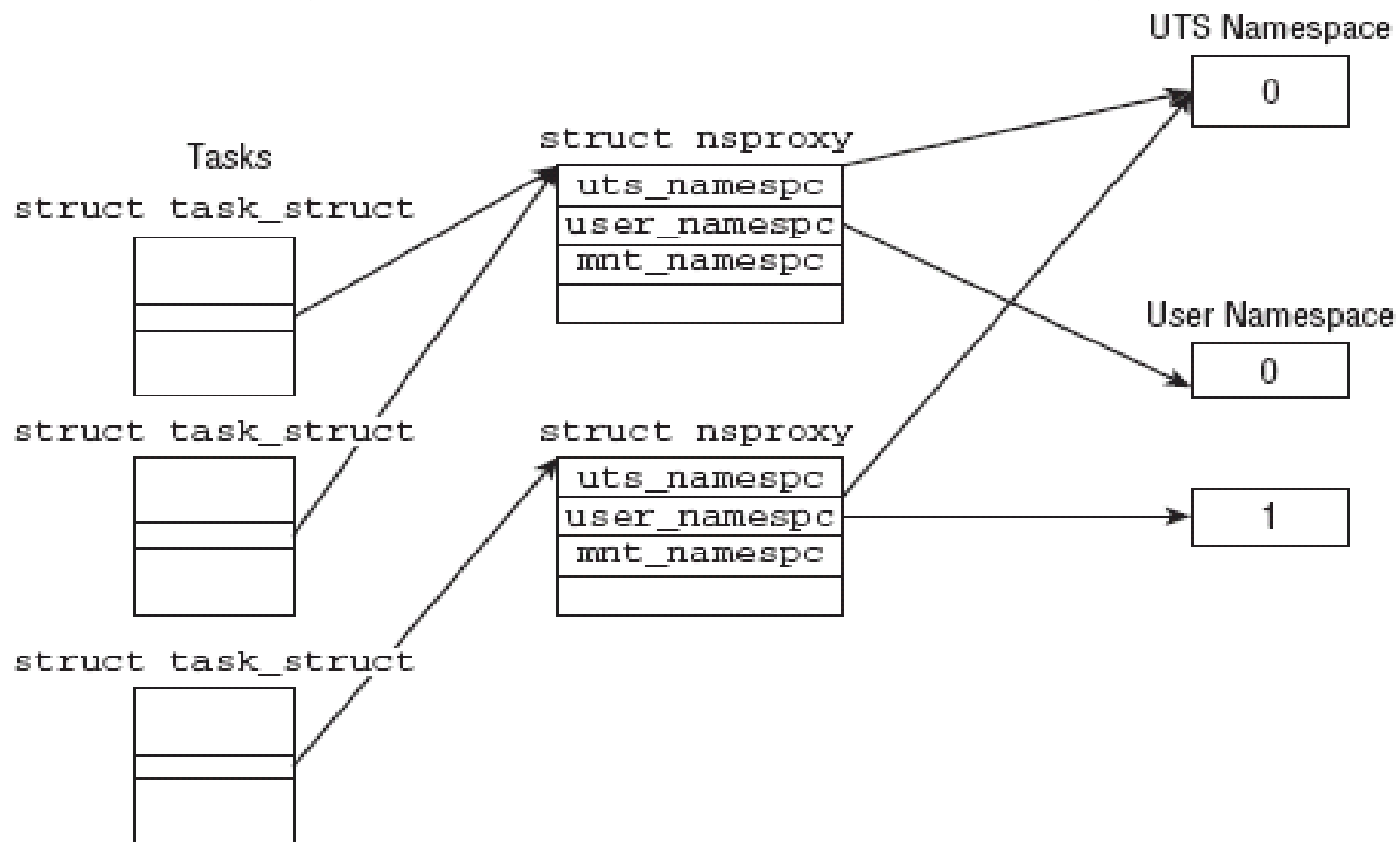
#define CLONE_NEWUSER 0x10000000

#define CLONE_NEWPID 0x20000000

define CLONE_NEWNET 0x40000000

实现

- 命名空间的实现需要两个部分
 - 内核子系统命名空间
 - 将此前所有的全局组件包装到命名空间中
 - 关联：将给定进程关联到所属各个命名空间



nsproxy

- **struct nsproxy**

- 用于汇集指向特定子系统的命名空间包装器的指针

```
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;  
    struct net          *net_ns;  
};
```

Cont.

- **UTS (UNIX Timesharing System)**
 - 内核的名称、版本、底层体系结构类型等信息
- **ipc_namespace**
 - 与进程间通信（IPC）有关的信息，隔离了 **SysV** 风格的进程间通信，如 **POSIX** 消息队列和进程间共享内存机制
- **mnt_namespace**
 - 已挂载的文件系统的信息，隔离文件系统的挂载点。进入新的 **mount namespace** 时会继承老 **mount namespace** 的挂载关系，但之后新 **namespace** 中挂载关系的改变不会对原有 **namespace** 造成影响
- **pid_namespace**
 - 有关进程ID的信息
- **user_namespace**
 - 用于限制每个用户资源使用的信息，与 **pid namespace** 类似，从 **uid** 的层面隔离容器中的用户
- **network namespace**
 - 所有网络相关的命名空间参数，在系统级别虚拟化网络栈；通过 **network namespace** 隔离后的进程可以有独立的路由表，端口空间。结合 **Docker** 提供的端口映射可以做主机网络到容器内的网络通信。
- **Cgroup namespace**
 - **4.6**加入的一个 **namespace**，用于隐藏宿主机上的 **cgroup** 信息，实现进程**cgroup**的隔离。每个**cgroupnamespace**有自己的一套**cgroup**根目录。使用 **CLONE_NEWCGROUP** 标识进入新 **namespace** 之后，其当前的**cgroup** 目录就成为了新 **namespace** 的 **cgroup** 根目录。

Cont.

- **init_nsproxy**

- 定义初始全局命名空间
- 指向各子系统初始的命名空间对象

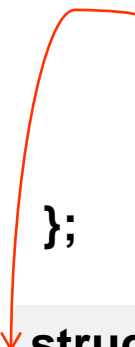
```
struct nsproxy init_nsproxy = {  
    .count = ATOMIC_INIT(1),  
    .uts_ns = &init_uts_ns,  
    #if defined(CONFIG_POSIX_MQUEUE) ||  
    defined(CONFIG_SYSVIPC)  
        .ipc_ns = &init_ipc_ns,  
    #endif  
    .mnt_ns = NULL,  
    .pid_ns = &init_pid_ns,  
    #ifdef CONFIG_NET  
        .net_ns = &init_net,  
    #endif  
};
```

UTS

- 所有相关信息都汇集到下列结构的一个实例中

<utsname.h>

```
struct uts_namespace {  
    struct kref kref;  
    struct new_utsname name;  
    struct user_namespace  
        *user_ns;  
};
```



```
struct new_utsname {  
    char sysname[65];  
    char nodename[65];  
    char release[65];  
    char version[65];  
    char machine[65];  
    char domainname[65];  
};
```

init/version.c

```
struct uts_namespace init_uts_ns = {  
    .kref = {  
        .refcount = ATOMIC_INIT(2),  
    },  
    .name = {  
        .sysname = UTS_SYSNAME,  
        .nodename = UTS_NODENAME,  
        .release = UTS_RELEASE,  
        .version = UTS_VERSION,  
        .machine = UTS_MACHINE,  
        .domainname = UTS_DOMAINNAME,  
    },  
    .user_ns = &init_user_ns,  
    .proc_inum = PROC_UTS_INIT_INO,  
};  
EXPORT_SYMBOL_GPL(init_uts_ns);
```

Cont.

- 创建
 - Use `CLONE_NEWUTS` in `fork()`
 - `copy_utsname()`
 - a copy of the previous instance of `uts_namespace`

pid_namespace

- **<pid_namespace.h>**

```
struct pid_namespace {  
    ...  
    struct task_struct *child_reaper;  
    ...  
    int level;  
    struct pid_namespace *parent;  
};
```

task_struct

- **<sched.h>**

```
struct task_struct {  
    ...  
    pid_t pid;  
    pid_t tgid;  
    ...  
    /* PID/PID hash table linkage. */  
    struct pid_link pids[PIDTYPE_MAX];  
    ...  
}
```

```
struct pid_link  
{  
    struct hlist_node node;  
    struct pid *pid;  
};
```

```
<pid.h>  
enum pid_type  
{  
    PIDTYPE_PID,  
    PIDTYPE_PGID,  
    PIDTYPE_SID,  
    PIDTYPE_MAX  
};
```

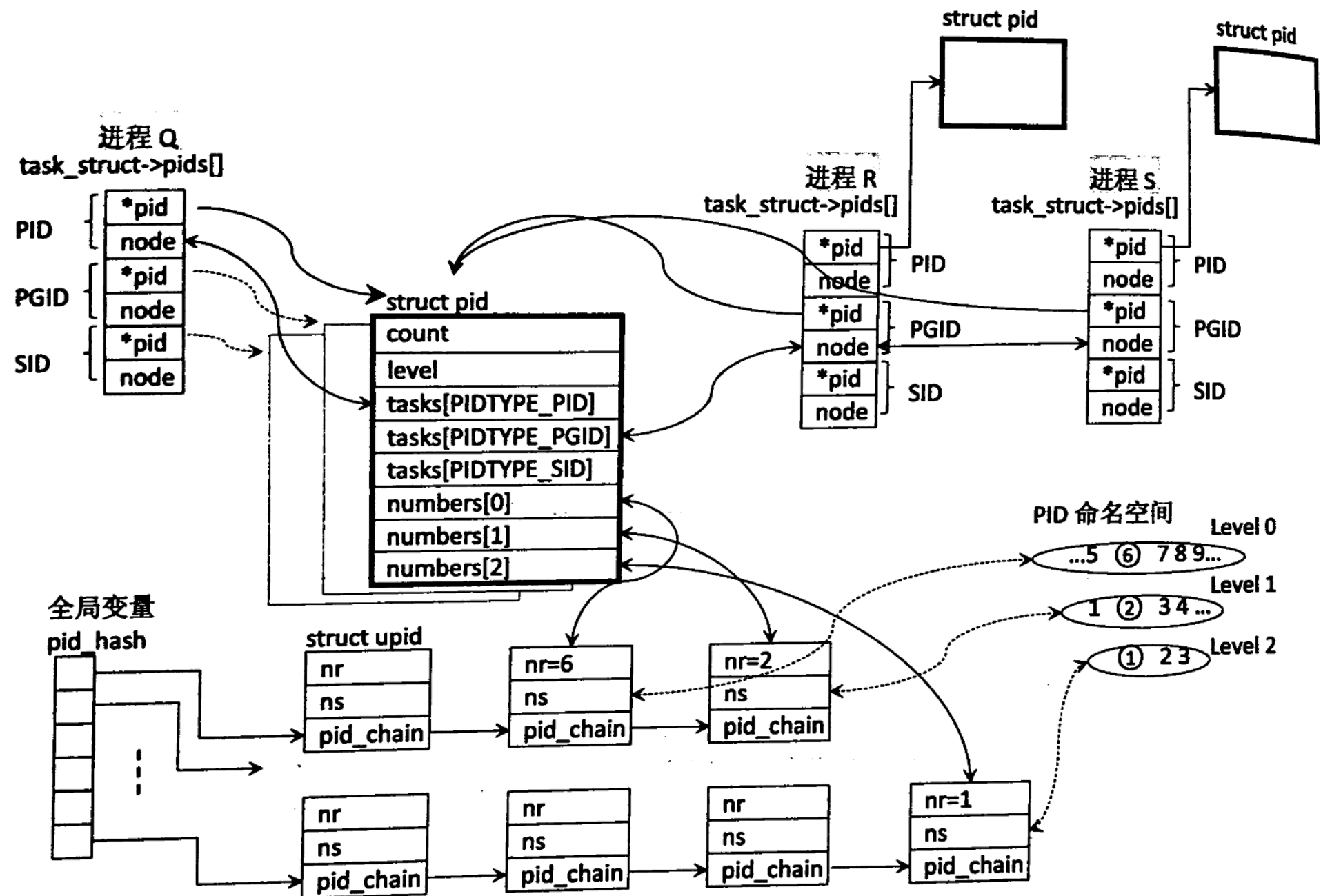
pid & upid

- **<pid.h>**

```
struct upid {  
    int nr;  
    struct pid_namespace *ns;  
};
```

```
struct pid  
{  
    atomic_t count;  
    /* lists of tasks that use this pid */  
    struct hlist_head tasks[PIDTYPE_MAX];  
    int level;  
    struct upid numbers[1];  
};
```

```
<pid.h>  
enum pid_type  
{  
    PIDTYPE_PID,  
    PIDTYPE_PGID,  
    PIDTYPE_SID,  
    PIDTYPE_MAX  
};
```

attach pid to task_struct

kernel/pid.c

```
void attach_pid(struct task_struct *task, enum
pid_type type, struct pid *pid)
```

```
{
```

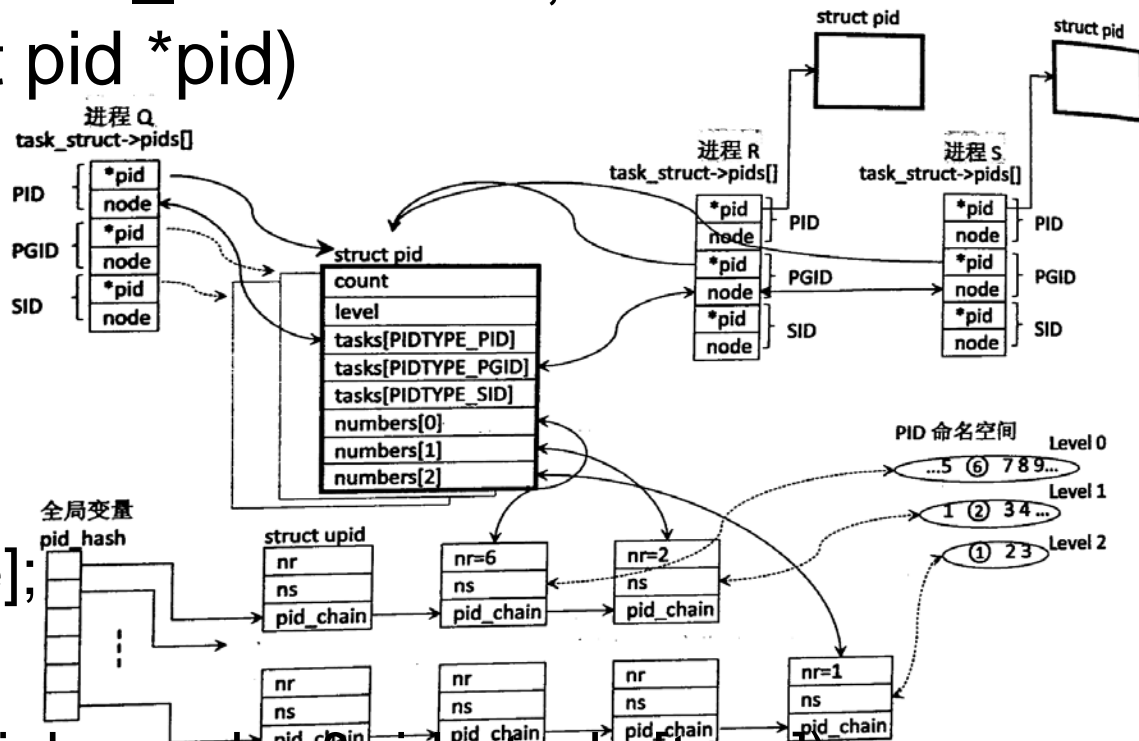
```
    struct pid_link *link;
```

```
    link = &task->pids[type];
```

```
    link->pid = pid;
```

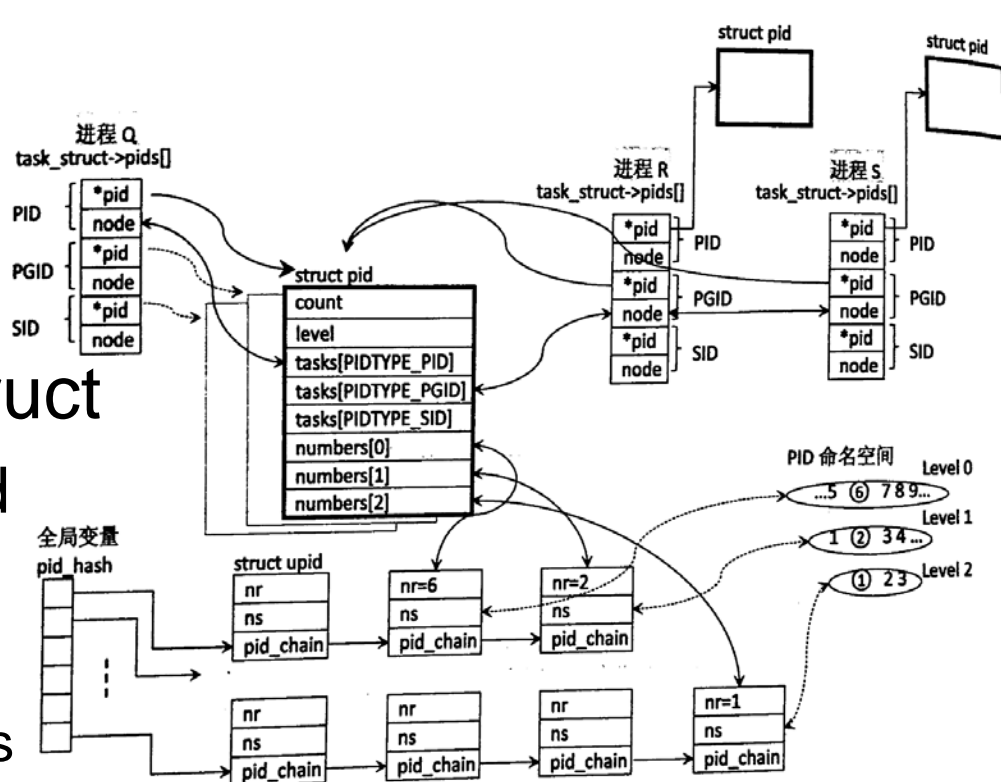
```
    hlist_add_head_rcu(&link->node, &pid->tasks[type]);
```

```
}
```



Functions

- Local ID+ns → task_struct
 - Local ID+ns → struct pid
 - find_pid_ns
 - struct pid → task_struct
 - find_task_by_pid_type_ns
- task_struct+ID type+ns → Local ID
 - Task_struct → struct pid
 - get_task_pid
 - Struct pid → Local ID
 - pid_nr_ns



Local pid 创建

kernel/pid.c

```
struct pid *alloc_pid(struct pid_namespace *ns)
{
    struct pid *pid;
    enum pid_type type;
    int i, nr;
    struct pid_namespace *tmp;
    struct upid *upid;
    ...
    tmp = ns;
    for (i = ns->level; i >= 0; i--) {
        nr = alloc_pidmap(tmp);
        ...
        pid->numbers[i].nr = nr;
        pid->numbers[i].ns = tmp;
        tmp = tmp->parent;
    }
    pid->level = ns->level;
    ...
}
```

Agenda

1. Process Management

– Introduction of Process

- What is process?
- Process and thread.

– Process in Linux

- Process descriptor
- Process creation
- Process Switch
- Process Termination

2. namespace

3. Process Scheduling

主要内容

- 进程的分类
- 调度策略
- 调度算法
- Linux 中的调度器
 - ✓ 2.4的 内核的调度器
 - ✓ O(1)调度器
 - ✓ CFS调度器

进程的分类

第一种分类:

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

CPU-bound

计算密集型

需要大量的CPU时间进行运算

第二种分类

交互式进程（interactive process）

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

批处理进程（batch process）

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

实时进程（real-time process）

有实时需求，不应被低优先级的进程阻塞

响应时间要短

典型的实时进程：视频/音频、机械控制等

实时进程的分类

- 硬实时进程（**hard real-time processes**）
 - 指定的时限内完成
 - 不一定是短时限
 - **Linux**内核默认不支持硬实时
- 软实时进程（**soft real-time processes**）
 - 硬实时进程的一种弱化形式
 - 尽快响应，**CPU**时间分配优先于普通进程
 - 如**CD**写入程序
- 普通进程（**normal processes**）
 - 优先级
 - 批处理（非交互、**CPU**使用密集，如编译、计算）低
 - 交互高

主要内容

- 进程的分类
- 调度策略
- 调度算法
- Linux 中的调度器
 - ✓ 2.4的 内核的调度器
 - ✓ O(1)调度器
 - ✓ CFS调度器

调度策略 (scheduling policy)

➤ What is Scheduling Policy?

- The set of rules used to determine **when and how** to select a new process to run is called the scheduling policy

➤ What is the objectives of a good Scheduling Policy?

- Fast process **response time**
- Good **throughput** for background jobs
- Avoidance of process **starvation**
- Etc.

Linux 的调度策略

- Linux scheduling is based on **time sharing**, CPU time is divided into slices, one for each runnable process
- If a currently running process is not terminated when its **quantum expires**, a **process switch** may take place
- The scheduling policy **ranks** processes according to their **priority**
- In Linux, process **priority is dynamic**.
 - Processes that have been denied the use of the CPU for a long time are boosted by dynamically increasing their priority
 - Long running processes have their priority lowered.

主要内容

- 进程的分类
- 调度策略
- 调度算法
- Linux 中的调度器
 - ✓ 2.4的 内核的调度器
 - ✓ $O(1)$ 调度器
 - ✓ CFS调度器

进程的切换时间多久合适？

- Neither too long nor too short
 - Too short: overhead for process switch
 - Too long: processes no longer appear to be executed concurrently
- Always a compromise
 - The rule of thumb: to choose a duration as long as possible, while keeping good system response time

在调度算法中哪些部分是最重要的？

- Minimize Response Time
 - Elapsed time to do an operation (job)
 - Response time is what the user sees
 - Time to echo keystroke in editor
 - Time to compile a program
 - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
 - Jobs per second
 - Throughput related to response time, but not identical
 - Minimizing response time will lead to more context switching than if you maximized only throughput
 - Minimize overhead (context switch time) as well as efficient use of resources (CPU, disk, memory, etc.)
- Fairness
 - Share CPU among users in some equitable way
 - Not just minimizing average response time

常见的基本调度算法

- 先来先服务算法(FCFS)
- 时间片轮转调度(RR)

先来先服务算法(FCFS)

- “Run until Done:” FIFO algorithm
- In the beginning, this meant one program runs non-preemptively until it is **finished** (including any blocking for I/O operations)
- Now, FCFS means that a process keeps the CPU until one or more threads **block**
- FCFS Scheme: Potentially bad for short jobs!
 - Depends on submit order
 - If you are first in line at the supermarket with milk, you don't care who is behind you; on the other hand...

时间片轮转调度(RR)

- Round Robin Scheme
 - Each process gets a small unit of CPU time (time quantum)
 - Usually 10-100 ms
 - After quantum expires, the process is preempted and added to the end of the ready queue
 - Suppose N processes in ready queue and time quantum is Q ms:
 - Each process gets $1/N$ of the CPU time
 - In chunks of at most Q ms
 - **What is the maximum wait time for each process?**
 - No process waits more than $(n-1)q$ time units

主要内容

- 进程的分类
- 调度策略
- 调度算法
- **Linux 中的调度器**
 - ✓ 2.4的 内核的调度器
 - ✓ $O(1)$ 调度器
 - ✓ CFS调度器

2.4内核的调度器

- 1400 行代码
- 3个基本的结构体组成。
- 最核心的结构体为schedule_data.
- 这个数据结构包含一个指针，指向当前运行的进程以及最后一次调度的时间戳
- 只有一个运行队列，它是由一个链表实现的。

Schedule_data

```
struct schedule_data {  
    struct task_struct * curr;  
    cycles_t last_schedule;  
} schedule_data;  
char __pad [SMP_CACHE_BYTES];  
static LIST_HEAD(runqueue head);
```

- Remarkably simple data structure
- Defined in sched.c
- Contains a time stamp of the last process switch
- Also contains a pointer to the process that is currently running

在 2.4 内核中，就绪进程队列是一个全局数据结构，调度器对它的所有操作都会因全局自旋锁而导致系统各个处理机之间的等待，使得就绪队列成为一个明显的瓶颈。

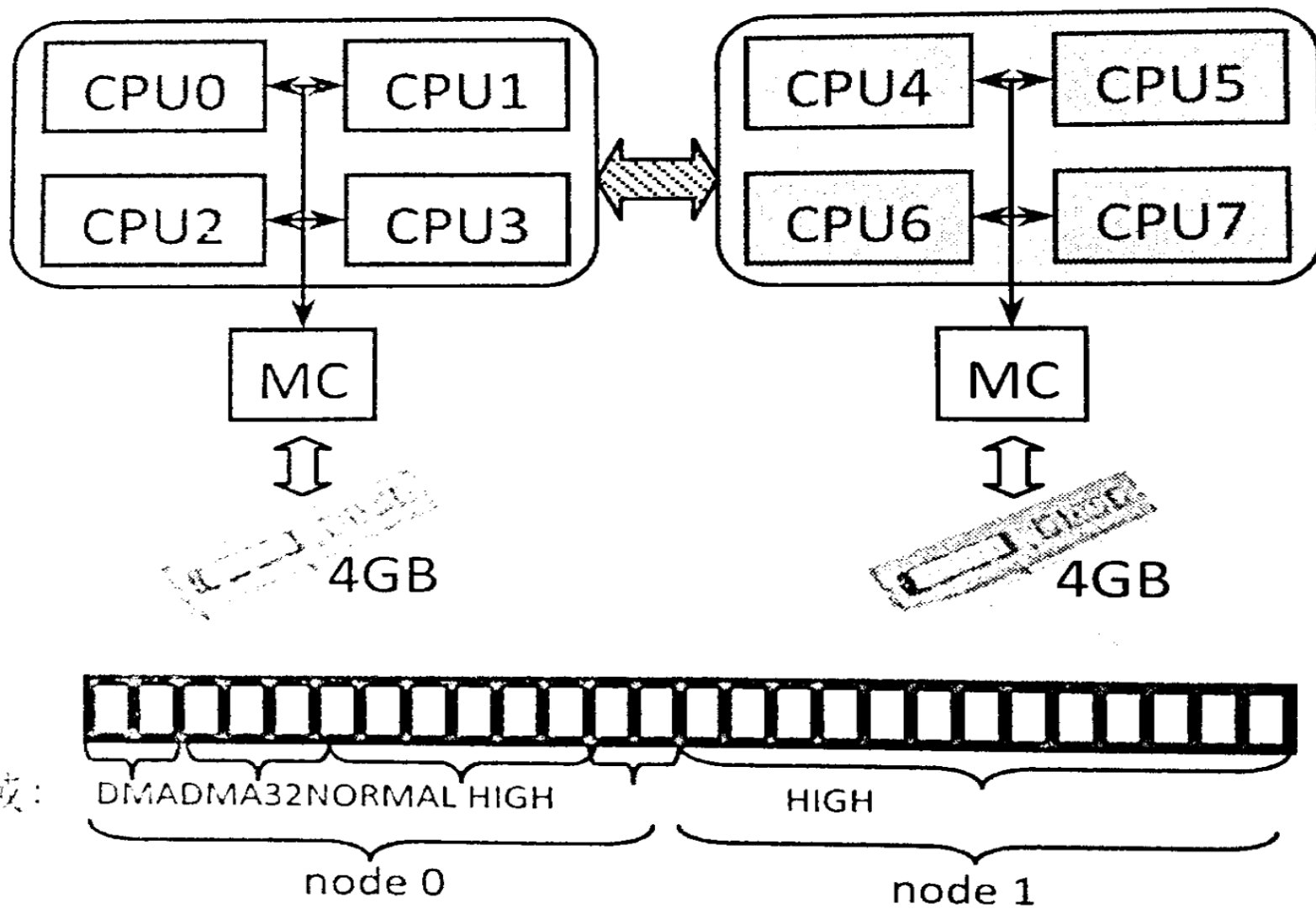
2.4 的就绪队列是一个简单的以 `runqueue_head` 为头的双向链表

在2.4内核中重新计算时间片是在所有就绪进程的时间片都用完以后才统一进行的，因而进程时间片的计算非常耗时

主要内容

- 进程的分类
- 调度策略
- 调度算法
- **Linux 中的调度器**
 - ✓ 2.4的 内核的调度器
 - ✓ **O(1)调度器**
 - ✓ CFS调度器

NUMA 架构



Linux 2.6 O(1) 调度系统

常数时间内完成工作(不依赖于系统上运行的进程数目)

2.6 O(1)调度系统从设计之初就把开发重点放在更好满足**实时性**和**多处理机并行性**上，并且基本实现了它的设计目标。

主要设计者**Ingo Molnar**将O(1)调度系统的特性概括为以下几点：

1) 继承和发扬 2.4 版调度器的特点

交互式作业优先

轻载条件下调度/唤醒的高性能

基于优先级调度

... ..

2) 在此基础之上的新特性:

- $O(1)$ 调度算法, 调度器开销恒定 (与当前系统负载无关), **实时性能更好**
- 全面**实现SMP的可扩展性**。每个处理器拥有自己的锁和自己的可执行队列。
- 新设计的**SMP亲和**方法, 尽量将**相关**一组任务分配给一个CPU进行连续的执行
- 加强**交互性能**。
- **保证公平**。在合理的时间范围内, 没有进程会处于饥饿状态。

在 $O(1)$ 中，就绪队列定义为一个复杂得多的数据结构 `struct runqueue`，并且尤为关键的是，**每一个CPU都将维护一个自己的就绪队列，这将大大减小竞争。**

$O(1)$ 算法中很多关键技术都与 `runqueue` 有关。

运行队列(runqueue)

- 调度中最基本的数据结构;
- 是给定处理器上的可运行进程的列表;
- 每个处理器一个;
- 每个可运行的进程唯一归属于一个runqueue;
- 包含每个处理器的调度信息;
- 定义在<kernel/sched.c>中。

结构runqueue

活动优先级队列
超时**优先级队列**
实际优先级数组

保护运行队列的**自旋锁**
可运行任务**数目**
队列最后被换出**时间**
当前运行任务
该处理器的**空任务**
.....

```

struct runqueue {
    spinlock_t lock;

    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    unsigned long nr_running;
#ifdef CONFIG_SMP
    unsigned long cpu_load[3];
#endif
    unsigned long long nr_switches;

    /*
     * This is part of a global counter where only the total sum
     * over all CPUs matters. A task can increase this counter on
     * one CPU and if it got migrated afterwards it may decrease
     * it on another CPU. Always updated under the runqueue lock:
     */
    unsigned long nr_uninterruptible;

    unsigned long expired_timestamp;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;

```

prio_array_t *active, *expired, arrays[2]

runqueue 中**最关键的数据结构**。

每个 CPU 的就绪队列按时间片是否用完分为两部分，

- active 指向时间片没用完、当前可被调度的就绪进程
- expired 指向时间片已用完的就绪进程。

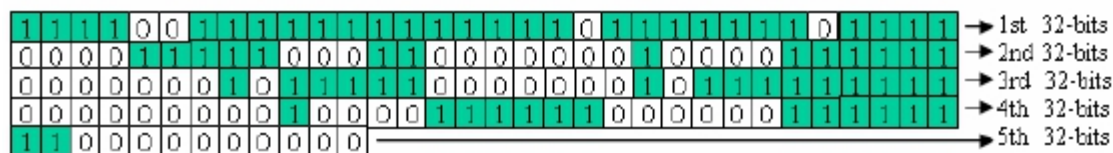
arrays 二元数组是**两类就绪队列的容器**，
active 和 expired 分别指向其中一个。

```
typedef struct prio_array prio_array_t;
```

运行队列(runqueue)

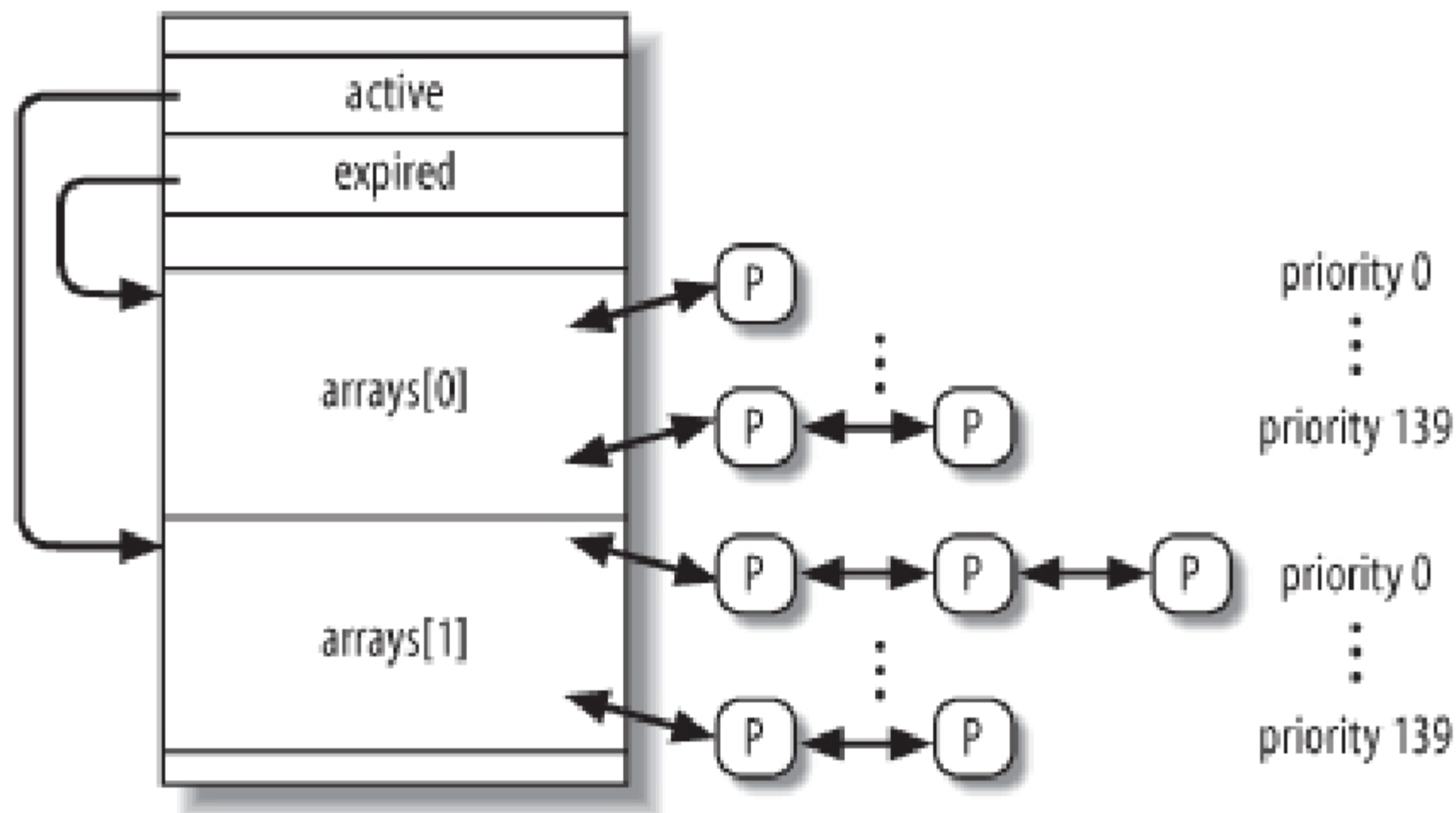
prio_array_t的结构如下:

```
struct prio_array {  
    int nr_active;      /*本进程组中进程个数*/  
    struct list_head queue[MAX_PRIO];  
                        /*每个优先级的进程队列*/  
    unsigned long bitmap[BITMAP_SIZE];  
                        /*上述进程队列的索引位图*/  
};
```

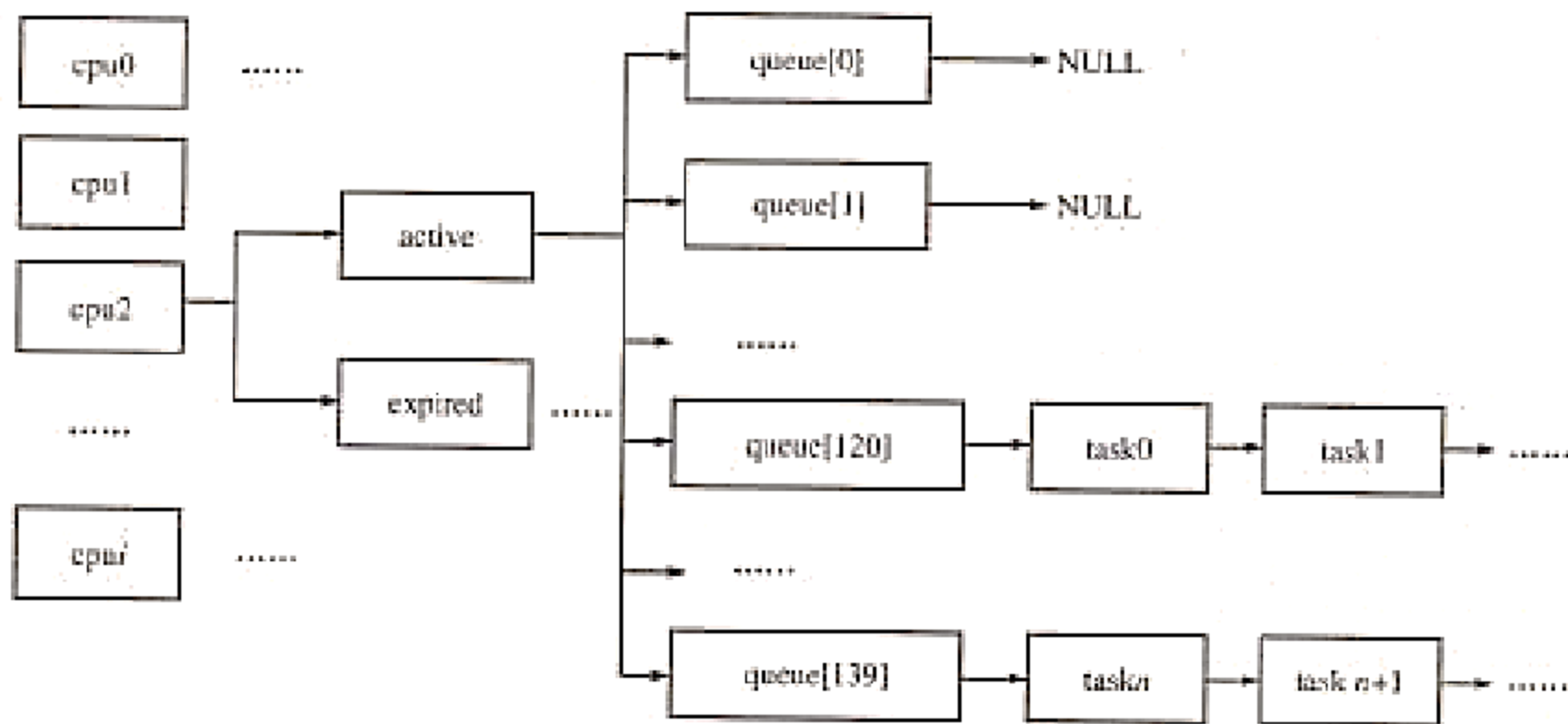


140-bits bitmap for priority array

```
#define BITMAP_SIZE (((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))
```

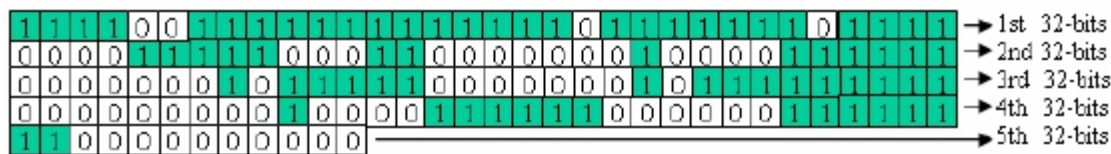



运行队列(runqueue)

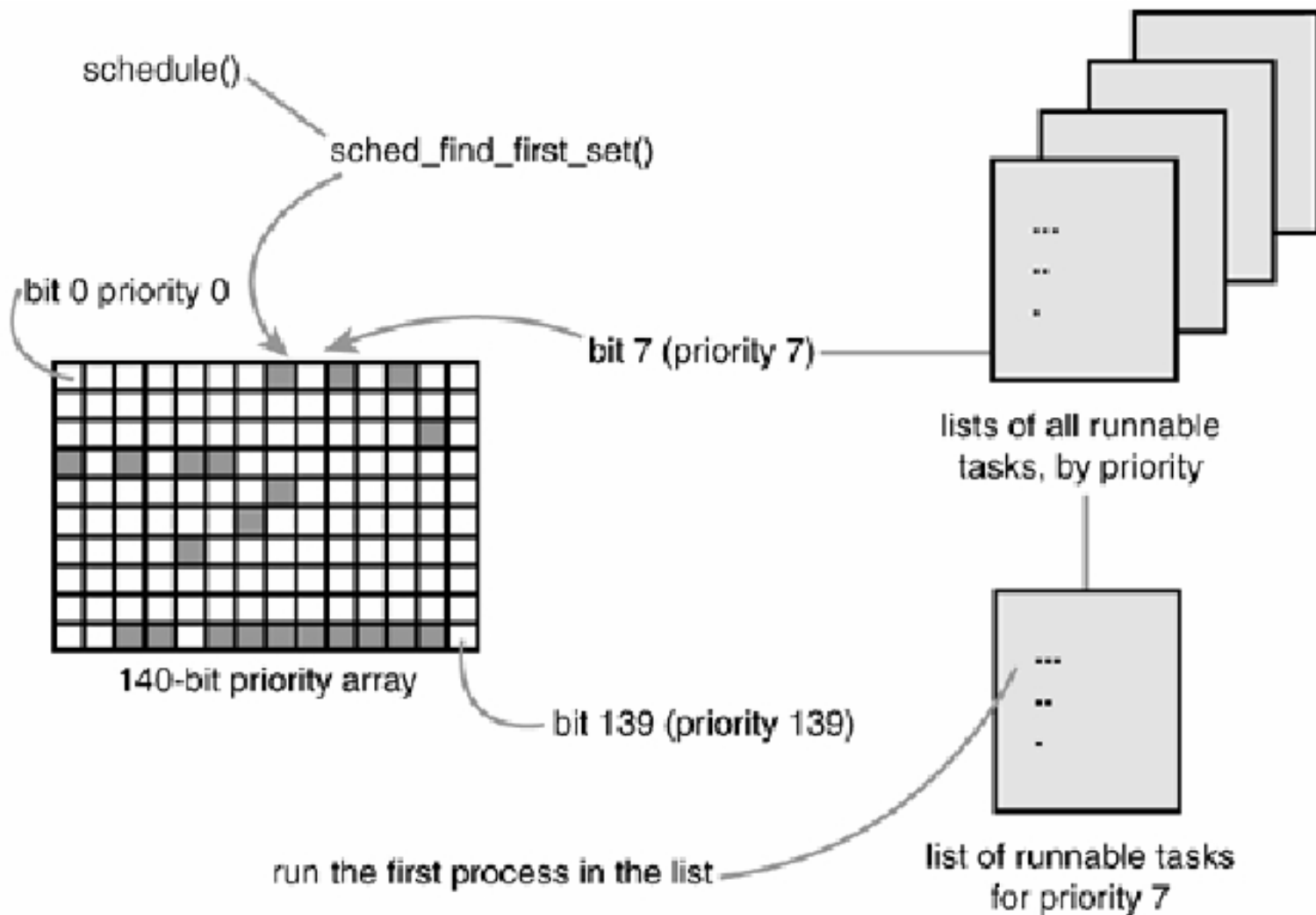


每个 cpu 的运行队列示意图

位图



140-bits bitmap for priority array



task_struct的改进

随着调度器的改进，`task_struct` 的内容也有了改进；

- 有的属性是新增加的
- 有的属性的值的含义发生了变化
- 有的属性仅仅是改了一下名字

运行队列(runqueue)

spinlock_t lock

runqueue的自旋锁，当对runqueue进行操作时需要锁住只影响一个C P U上的就绪队列

task_t *curr

本C P U当前运行的进程。

进程描述符相关

struct thread_info *thread_info

当前进程运行的一些环境信息。其中有两个结构成员非常重要，与调度密切相关：

__s32 preempt_count;
unsigned long flags;

- **preempt_count**是用来表示**内核能否被抢占的使能成员**
 - 如果它大于0，表示内核不能被抢占；
 - 如果等于0，则表示内核处于安全状态（即没有加锁），可以抢占
- **flags**里面有一个**TIF_NEED_RESCHED**位
 - 如果此标志位为1，则表示应该尽快启动调度器。
 - 活动进程设置该标志，则调度器将从该进程收回CPU并授予新进程
 - 可以自愿或强制设置

进程描述符相关

int prio

- **prio**是进程的**动态优先级**，相当于Kernel2.4中用goodness()函数计算出来的结果；
- 在O(1)中不再是由调度器统一计算，而是独立计算；
- **0~MAX_PRIO-1**：数值越大优先级越小

动态优先级独立计算

- 非实时进程：取决于static_prio和sleep_avg
- 实时进程：rt_priority，由sys_sched_setschedule()设置

优先级的计算

进程优先级范围是从0 ~ MAX_PRIO-1,

- 实时进程的优先级的范围是0 ~ MAX_RT_PRIO-1
- 普通进程的优先级是MAX_RT_PRIO ~ MAX_PRIO-1
- 数值越小优先级越高

```
#define MAX_USER_RT_PRIO      100
#define MAX_RT_PRIO          MAX_USER_RT_PRIO

#define MAX_PRIO              (MAX_RT_PRIO + 40)
```


进程描述符相关

int static_prio

static_prio则是进程的**静态优先级**，与nice意义相同。nice的取值仍然是-20 ~ 19，数值越小，进程优先级越高。

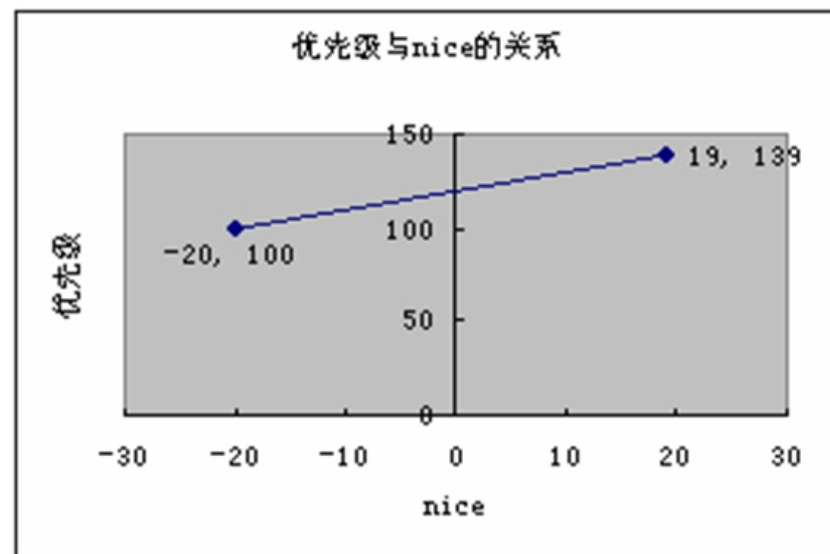
kernel/sched/sched.h中定义了两个宏：

```
#define NICE_TO_PRIO(nice)    (MAX_RT_PRIO + (nice) + 20)
```

```
#define PRIO_TO_NICE(prio)    ((prio) - MAX_RT_PRIO - 20)
```

可见priority和nice的关系是：

$\text{priority} = \text{MAX_RT_PRIO} + \text{nice} + 20$



进程描述符相关

unsigned long sleep_avg

- **进程的平均等待时间**
- 当进程处于等待或者睡眠状态时，该值变大；当进程运行时，该值变小。

值越大，计算出来的进程优先级？

- sleep_avg是O(1)调度中衡量进程的一个关键指标，它既可以**用来衡量进程的交互程度**，也可以用来**衡量进程的紧急程度**。

进程描述符相关

unsigned long policy

进程的**调度策略**和2.4一样，有以下几种：

SCHED_FIFO 先进先出式调度，除非有更高优先级进程申请运行，否则该进程将保持运行至退出才让出CPU；

SCHED_RR 轮转式调度，该进程被调度下来后将被置于运行队列的末尾，以保证其他实时进程有机会运行

SCHED_NORMAL 常规的优先级调度策略

```
#define SCHED_NORMAL 0
```

```
#define SCHED_FIFO 1
```

```
#define SCHED_RR 2
```

进程描述符相关

unsigned int time_slice, first_time_slice

time_slice是进程剩余的时间片

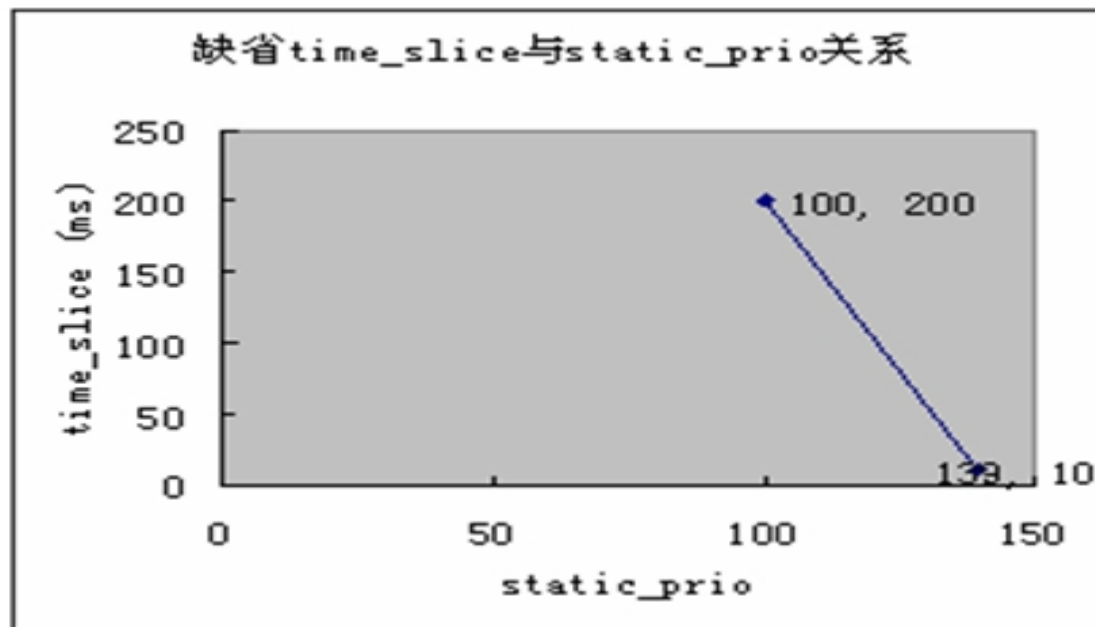
- 进程的默认时间片与进程的静态优先级相关
- 进程创建时，与父进程平分时间片
- 运行过程中递减，一旦归零，则重置时间片，并请求调度
递减和重置在时钟中断中进行（scheduler_tick()）
- 进程退出时，如果自身并未被重新分配时间片，则将自己剩余的时间片返还给父进程

first_time_slice用来记录时间片是否是第一次分配（进程创建时），以确定进程退出时是否将时间片交还给父进程。

计算时间片

- 进程的**时间片time_slice**是基于**进程静态优先级的**，静态优先级越高（值越小），时间片就越大。计算时间片是通过函数 `task_timeslice()`（`kernel/sched.c`）来完成的。
- 通过优先级来计算时间片的等式为：**

$$\text{timeslice} = \text{MIN_TIMESLICE} + ((\text{MAX_TIMESLICE} - \text{MIN_TIMESLICE}) * (\text{MAX_PRIO} - 1 - (\text{p}) \rightarrow \text{static_prio}) / (\text{MAX_USER_PRIO} - 1))$$



将 100~139 的优先级映射到 200ms~10ms 的时间片上
优先级数值越大，则分配的时间片越小。

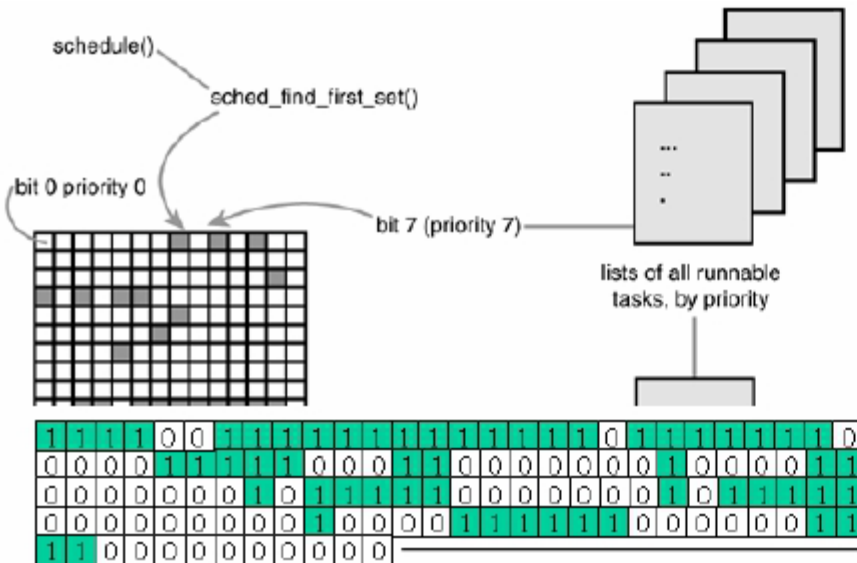
优先级的计算

以下几种情况需要计算进程的优先级：

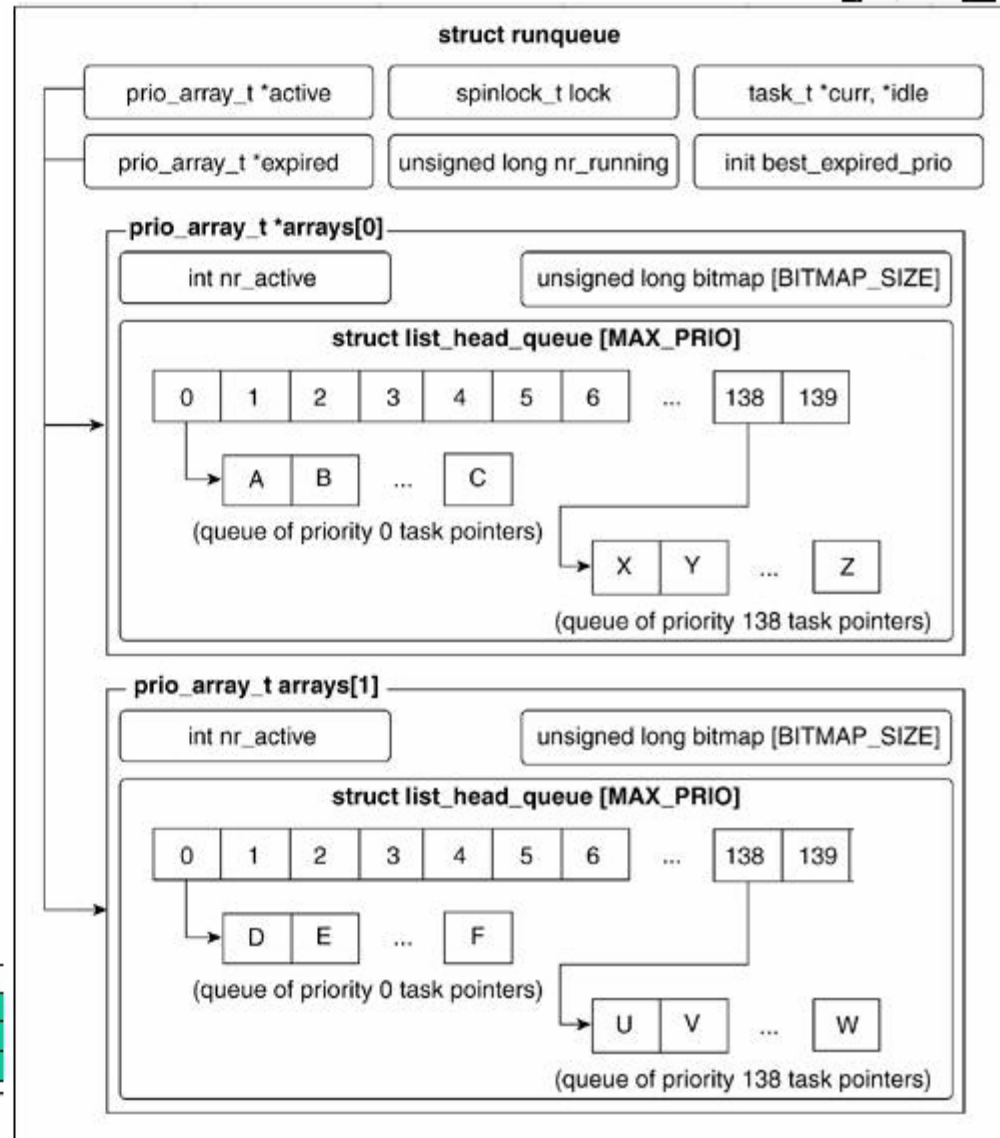
- 创建新进程，使用函数`effective_prio()`；
- 唤醒等待进程时，使用函数`recalc_task_prio()`来计算进程动态优先级。
- 进程用完时间片以后，被重新插入到`active array`或者`expired array`的时候需要重新计算动态优先级，以便将进程插入到队列的相应位置。此时，使用函数`effective_prio()`；
- 其他情况，如负载平衡（`move_task_away()`）以及修改 `nice` 值（`set_user_nice()`）、修改调度策略（`setscheduler()`）时。

Priority Arrays in a Run Queue

```
struct prio_array{
    int nr_active; /*任务数目*/
    unsigned long
        bitmap[BITMAP_SIZE];
    struct list_head
        queue[MAX_PRIO];
}
```

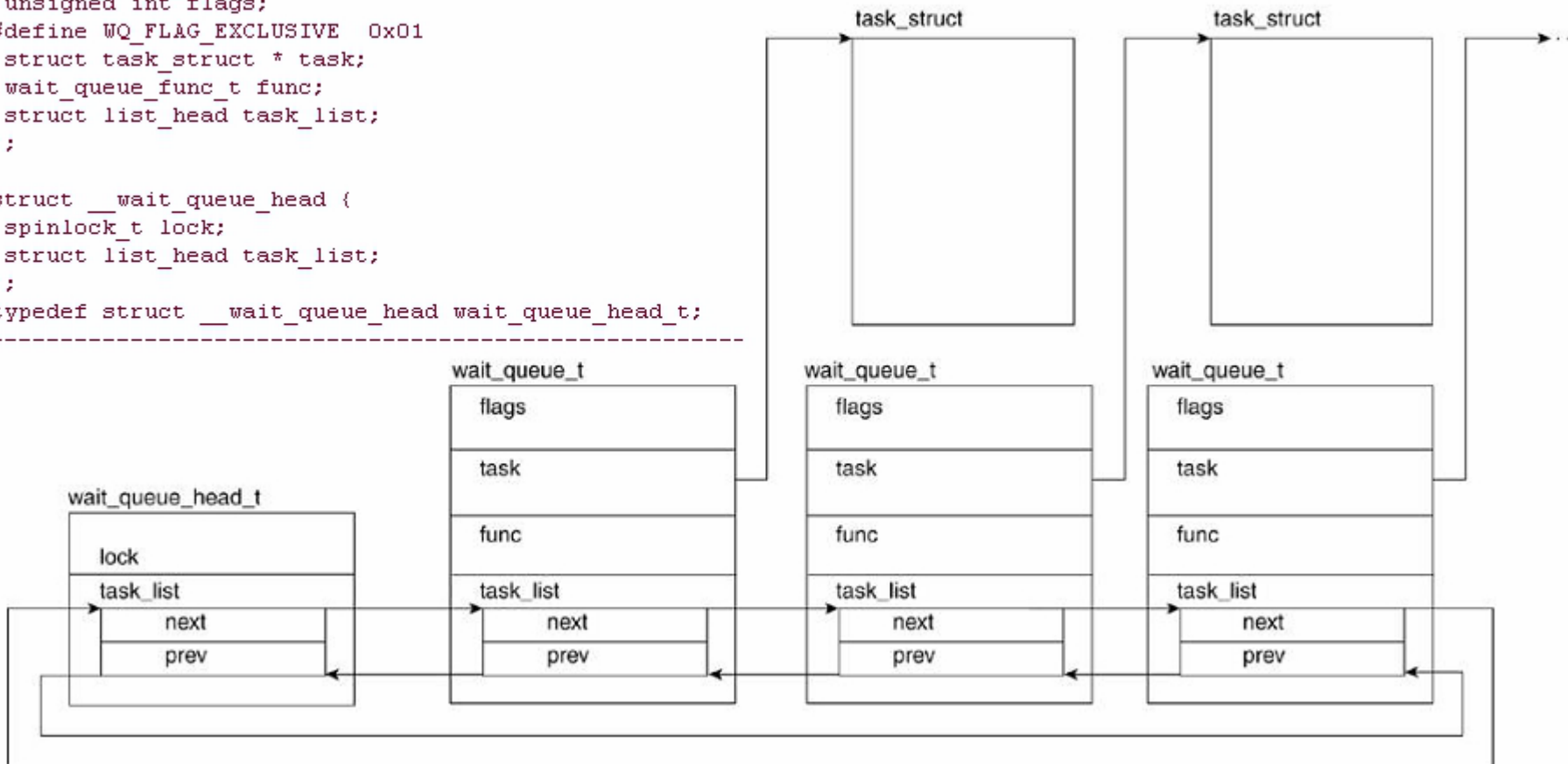


140-bits bitmap for priority array



Wait Queue Structures

```
include/linux/wait.h
19 typedef struct __wait_queue wait_queue_t;
...
23 struct __wait_queue {
24     unsigned int flags;
25     #define WQ_FLAG_EXCLUSIVE 0x01
26     struct task_struct * task;
27     wait_queue_func_t func;
28     struct list_head task_list;
29 };
30
31 struct __wait_queue_head {
32     spinlock_t lock;
33     struct list_head task_list;
34 };
35 typedef struct __wait_queue_head wait_queue_head_t;
```



Schedule调度函数何时被调用？

主动启动

当前进程因等待资源而需进入被阻塞状态时，调度程序将：

- 首先把当前进程放到适当的等待队列里
- 把当前进程的状态设为TASK_INTERRUPTIBLE或者TASK_UNINTERRUPTIBLE
- 调用schedule()，让新的进程运行

被动调度

- **通过设置当前进程的TIF_NEED_RESCHED标志位为1来请求调度**，设置时机：
 - 当前进程用完了它的CPU时间片，update_process_times()重新进行计算
 - 当一个进程被唤醒，而且它的优先级比当前进程高
 - 当sched_setschedler()或sched_yield()系统调用被调用时
- **每次**返回核心态（注意时钟中断）或者用户态，都可能检查**TIF_NEED_RESCHED**标志位的值，决定是否调用函数schedule()

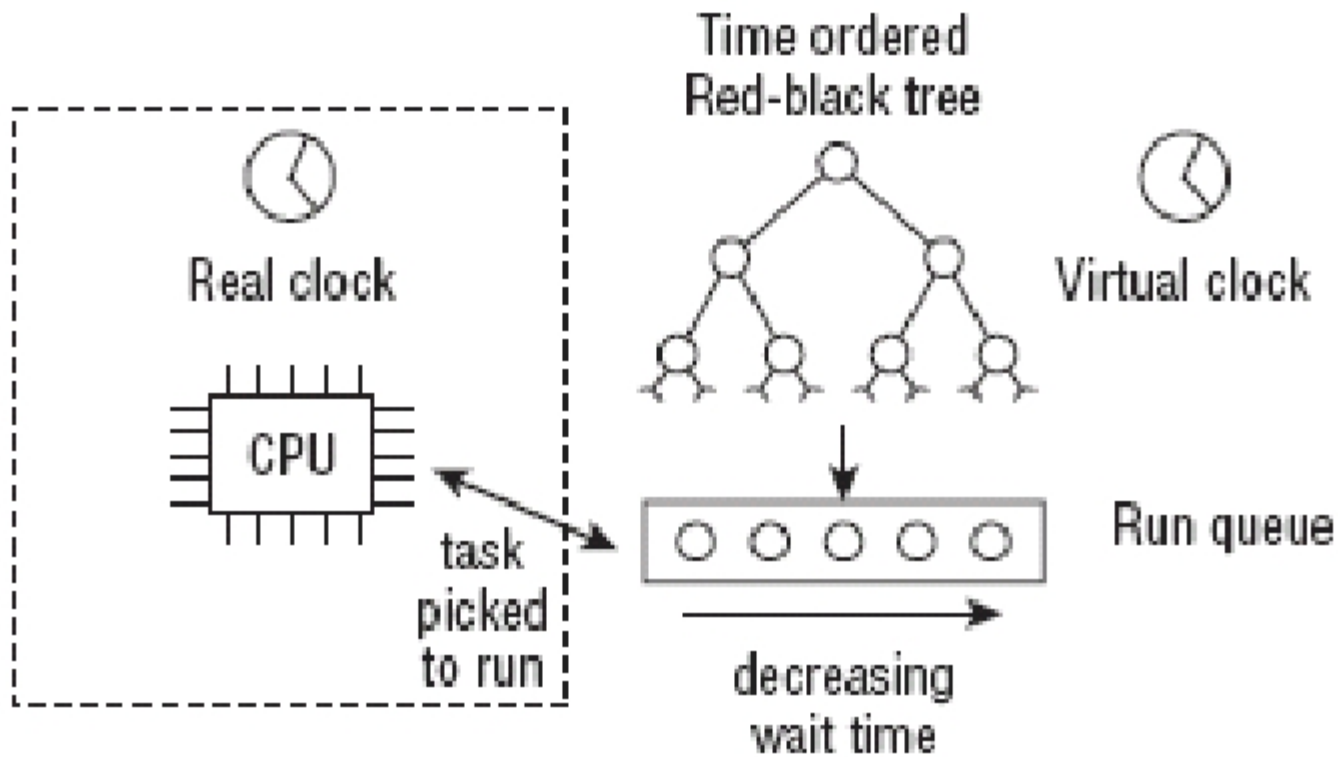
主要内容

- 进程的分类
- 调度策略
- 调度算法
- **Linux 中的调度器**
 - ✓ 2.4的 内核的调度器
 - ✓ **O(1)调度器**
 - ✓ **CFS调度器**

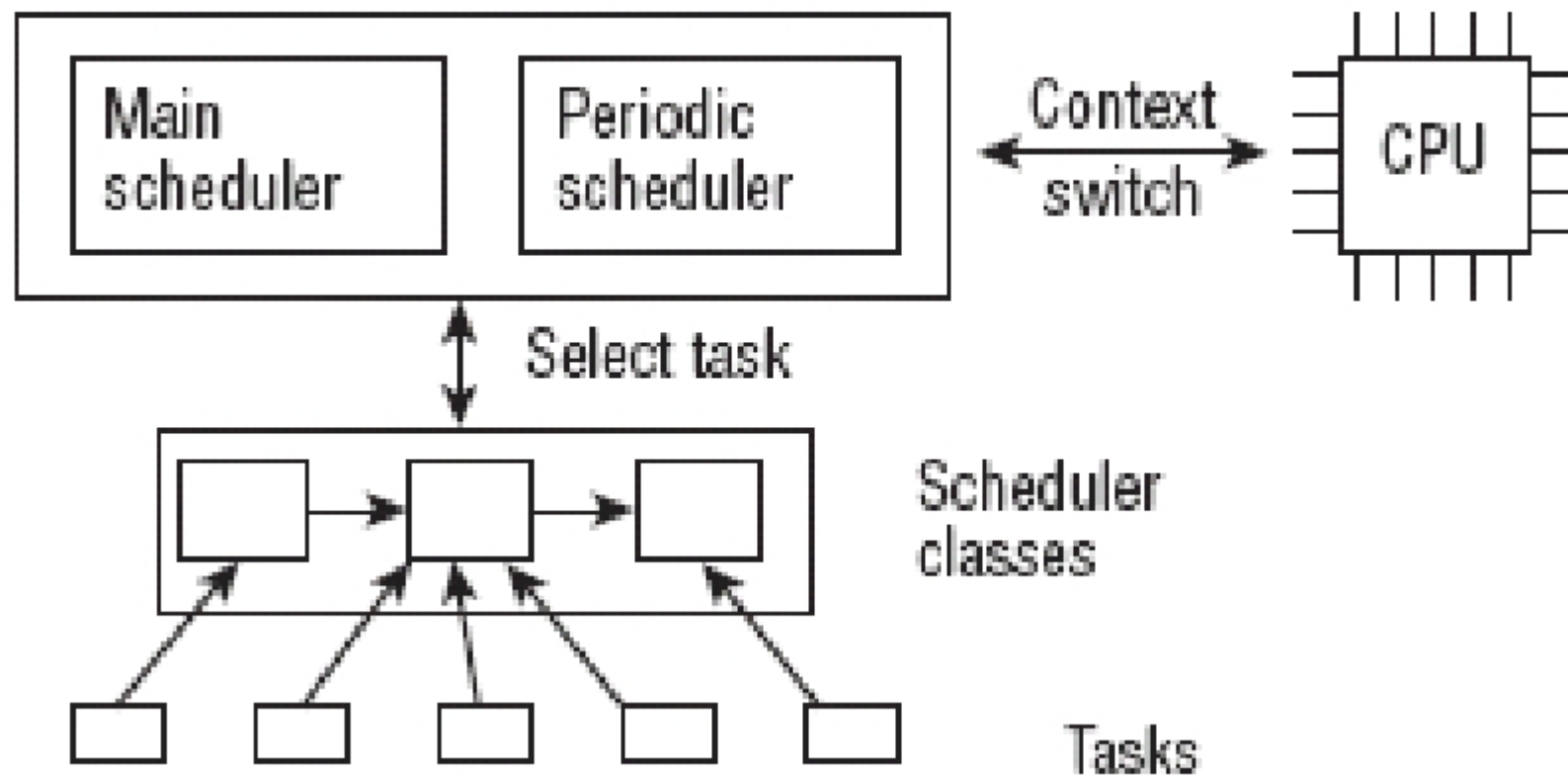
完全公平调度算法

Completely Fair Schedule

虚拟时钟



- 慢于实际的时钟
- 依赖于当前等待调度器挑选的进程的数目



数据结构

数据结构

<sched.h>

```
struct task_struct {  
    ...  
    int prio, static_prio, normal_prio;  
    unsigned int rt_priority;  
    struct list_head run_list;  
    const struct sched_class *sched_class;  
    struct sched_entity *se; ;  
    unsigned int policy;  
    cpumask_t cpus_allowed;  
    unsigned int time_slice;  
    ...  
}
```

task_struct

- 优先级
 - **static_prio**: 静态优先级
 - 进程启动时分配
 - 在进程运行期间保持恒定
 - 可以用**nice**和**sched_setscheduler()**系统调用修改
 - **normal_prio**: 动态优先级
 - 基于进程的静态优先级和调度策略计算出的优先级
 - **fork**继承**normal prio**
 - **prio**: 调度器考虑的优先级
 - 出于某些原因发生的临时改变会被计算进来
 - **rt_priority**: 实时进程的优先级
 - 大值代表高优先级

task_struct

Cont.

- **sched_class**: 该进程所属的调度器类。
- **sched_entity**: 可调度实体
 - 为了调度比进程更大的实体
 - 如组调度: CPU时间→组间分配→组内分配
 - 在进程注册到就绪队列时, **on_rq**成员为1, 否则为0
- **cpus_allowed**: 位域, 在多处理器系统上使用, 标示进程可以在哪些CPU上运行

task_struct Cont.

- Policy
 - SCHED_NORMAL
 - 普通进程
 - CFS
 - SCHED_RR、SCHED_FIFO
 - 软实时进程
 - 实时调度器类

kernel/sched/sched.h

 - static inline int rt_policy(int policy): 调度策略是否属于实时类。
 - static inline int task_has_rt_policy(struct task_struct *p): 进程是否使用实时调度策略
 - SCHED_DEADLINE
 - EDF调度策略的实时进程
 - dl_sched_class 调度器类
 - dl_policy(): 调度策略是否属于dl。
 - SCHED_IDLE
 - 相对权重最小
 - 不负责调度空闲进程（空闲进程由内核提供单独的机制来处理）
 - SCHED_BATCH
 - 不抢占CFS调度的进程（所以也不会抢占交互式进程）

task_struct

Cont.

- **run_list和time_slice:**
 - **SCHED_RR**
 - **run_list:** 表头，进程运行列表
 - **time_slice:** 指定进程可使用**CPU**的剩余时间片

Sched_class

```
struct sched_class {  
    const struct sched_class *next;  
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);  
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);  
    void (*yield_task) (struct rq *rq);  
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);  
    struct task_struct * (*pick_next_task) (struct rq *rq);  
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);  
    void (*set_curr_task) (struct rq *rq);  
    void (*task_tick) (struct rq *rq, struct task_struct *p);  
    void (*task_new) (struct rq *rq, struct task_struct *p);  
};
```

include/linux/sched.h

- **pick_next_task**时按照优先次序遍历调度类（处理相应的调度队列）：
stop_sched_class -> dl_sched_class -> rt_sched_class -> fair_sched_class -> idle_sched_class
- 编译时建立顺序

Cont.

yield_task

- 进程自愿放弃CPU
- 调用sched_yield系统调用

check_preempt_curr

- 如果条件允许，用一个新唤醒的进程来抢占当前进程
- 如用wake_up_new_task唤醒新进程时，会调用该函数判断被唤醒进程是否抢占当前进程

pick_next_task

- 选择下一个将要运行的进程，放到cpu上

put_prev_task

- 用另一个进程代替当前运行的进程之前调用，将当前进程从cpu上撤下

set_curr_task

- 在进程的调度策略发生变化时调用

task_tick

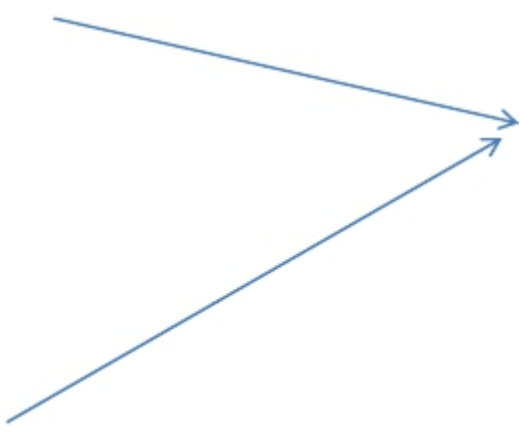
- 每次激活周期性调度器时，由周期性调度器调用

new_task

- 用于建立fork系统调用和调度器之间的关联。新进程创建后用其通知调度器

Cont.

- **fair_sched_class**
 - SCHED_NORMAL
 - SCHED_BATCH
 - SCHED_IDLE
- **rt_sched_class**
 - SCHED_RR
 - SCHED_FIFO
- **dl_sched_class**
 - SCHED_DEADLINE



**User space
applications**

Can not see schedule
classes, but can set
schedule policies

sched_entity

<sched.h>

```
struct sched_entity {  
    struct load_weight load; /* 用于负载均衡 */  
    struct rb_node run_node;  
    unsigned int on_rq;  
    u64 exec_start;  
    u64 sum_exec_runtime;  
    u64 vruntime;  
    u64 prev_sum_exec_runtime;  
    ...  
}
```

struct sched_rt_entity

struct sched_dl_entity

Cont.

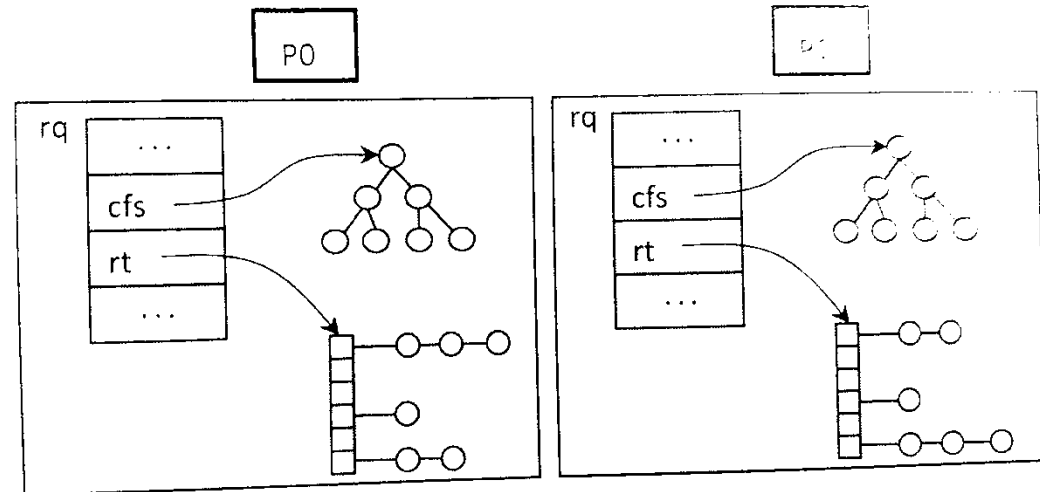
- **Load**
 - 权重，各实体占队列总负荷的比例
 - 计算负荷权重是调度器的一项重任, 因为CFS所需的虚拟时钟的速度最终依赖于负荷
- **run_node**
 - 红黑树结点，调度队列利用红黑树排序
- **on_rq**
 - 该实体当前是否在就绪队列
- **sum_exec_runtime**
 - 记录进程运行消耗的CPU时间
 - 跟踪运行时间是由**update_curr**不断累积完成的。调度器中许多地方都会调用该函数，例如，新进程加入就绪队列时，或者周期性调度器中。每次调用时，会计算当前时间和**exec_start**之间的差值，**exec_start**则更新到当前时间，差值则被加到**sum_exec_runtime**
 - 在进程被撤下CPU时，其当前**sum_exec_runtime**值保存到**prev_exec_runtime**(不重置**sum_exec_runtime**，**单调增长**)。此后，在进程抢占时又需要该数据
- **vruntime**
 - 进程执行期间虚拟时钟上流逝的时间

Run queues

- 1Rq / 1cpu Runqueues: per cpu 数组
- 1 Process → 1 rq

kernel/sched/sched.h

```
struct rq {  
    unsigned long nr_running;  
    #define CPU_LOAD_IDX_MAX 5  
    unsigned long  
    cpu_load[CPU_LOAD_IDX_MAX];  
    ...  
    struct load_weight load;  
    struct cfs_rq cfs;  
    struct rt_rq rt;  
    struct dl_rq dl;  
    struct task_struct *curr, *idle, *stop;  
    u64 clock;  
    ...  
};
```



- **nr_running**: 队列上可运行进程的数目
- **Load**: 度量就绪队列当前负荷
 - 队列的负荷，本质上与队列上当前活动进程的数目成正比，其中的各个进程又有优先级作为权重。每个就绪队列的虚拟时钟的速度即基于该信息
- **cpu_load**: 数组，跟踪此前的负荷状态
- **cfs**、**rt** 和 **dl**: 嵌入的子就绪队列，分别用于完全公平调度器和实时调度器
- **curr**: 指向当前运行的进程的**task_struct**实例
- **Idle**: 指向空闲进程的**task_struct**实例
- **clock**和**prev_raw_clock**
 - 用于实现就绪队列自身的时钟
 - 每次调用周期性调度器时，都会更新**clock**的值
 - 另外内核还提供了标准函数**update_rq_clock**，可在操作就绪队列的调度器中多处调用，例如，在用**wakeup_new_task**唤醒新进程时

Cont.

kernel/sched/sched.h

#define cpu_rq(cpu) (&per_cpu(runqueues, (cpu)))

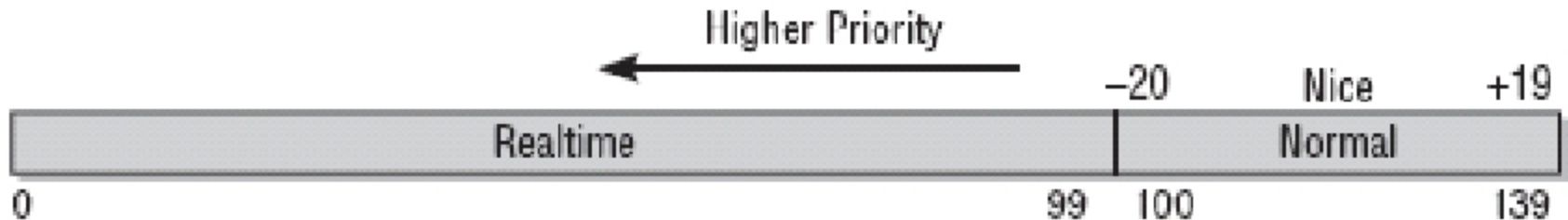
#define this_rq() (&__get_cpu_var(runqueues))

#define task_rq(p) cpu_rq(task_cpu(p))

define cpu_curr(cpu) (cpu_rq(cpu)->curr)

优先级设置

优先级定义



`<sched.h>`

```
#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)
```

`kernel/sched/sched.h`

```
#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)
```

`task_struct->prio / task_struct->normal_prio / task_struct-> static_prio`

优先级计算

```
p->prio = effective_prio(p);
```

```
kernel/sched/core.c
```

```
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)) //checks if the normal priority is in the real-time range
        return p->normal_prio;
    return p->prio;
}
```

Cont.

kernel/sched/core.c

```
static inline int normal_prio(struct task_struct *p)  
{  
    int prio;  
    if (task_has_rt_policy(p)) //scheduling policy set in the task_struct  
        prio = MAX_RT_PRIO-1 - p->rt_priority;  
    else  
        prio = __normal_prio(p);  
    return prio;  
}
```

```
kernel/sched/core.c  
static inline int __normal_prio(struct task_struct *p)  
{  
    return p->static_prio;  
}
```

Cont.

Task type / priority	<code>static_prio</code>	<code>normal_prio</code>	<code>prio</code>
Non-real-time task	<code>static_prio</code>	<code>static_prio</code>	<code>static_prio</code>
Priority-boosted non-real-time task	<code>static_prio</code>	<code>static_prio</code>	prio as before
Real-time task	<code>static_prio</code>	<code>MAX_RT_PRIO-1-rt_priority</code>	prio as before

effective_prio()设置prio:

- 在新建进程用**wake_up_new_task**唤醒时
- 使用**nice**系统调用改变静态优先级时

创建新进程:

- 子进程的静态优先级继承自父进程
- 子进程的动态优先级（**task_struct->prio**），设置为父进程的**normal_prio**

权重计算

进程权重: `task_struct->se.load`

```
<sched.h>
struct load_weight {
    unsigned long
    weight, inv_weight;
};
```

kernel/sched/sched.h

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

`prio_to_wmult[]`

```

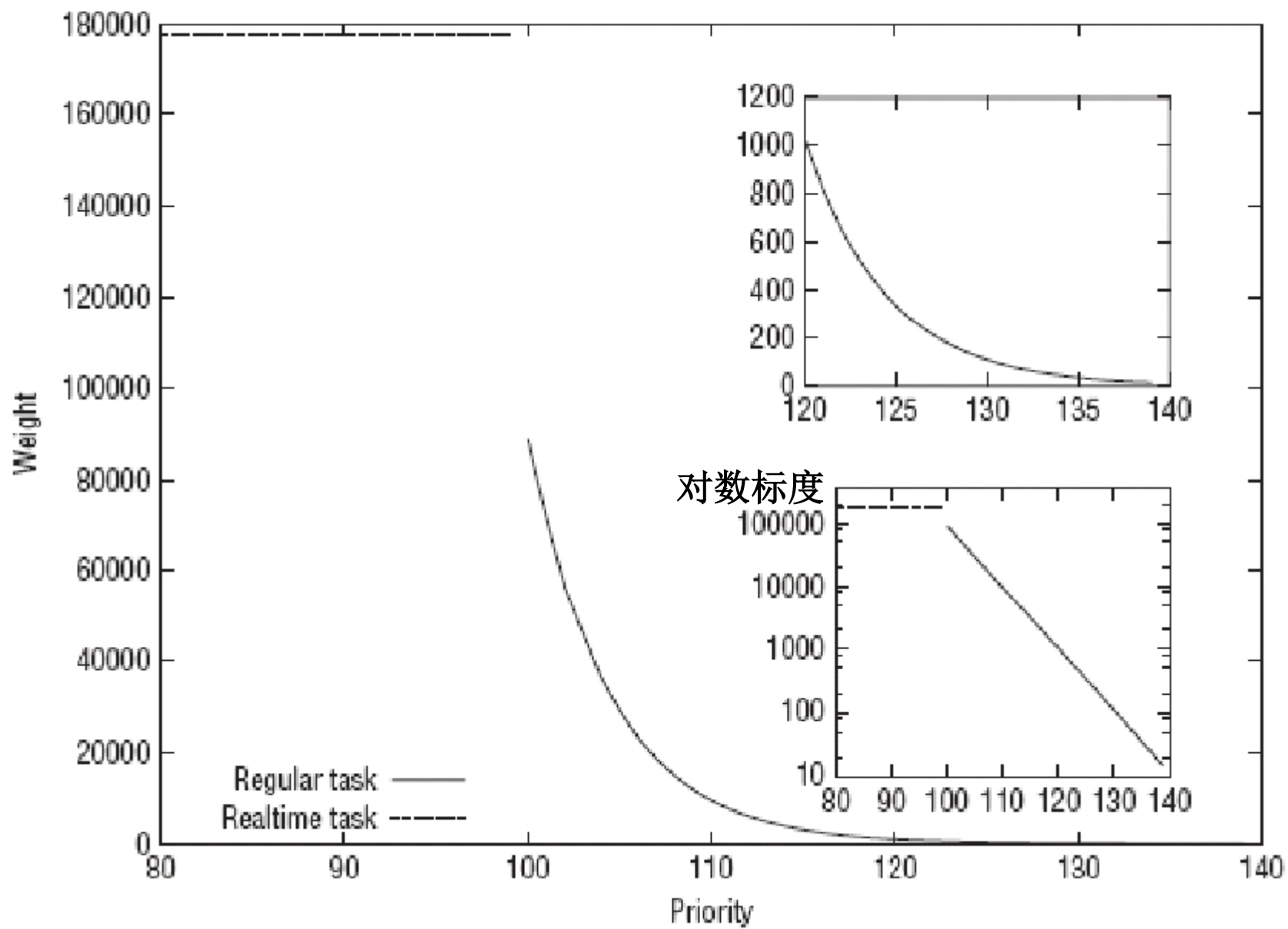
737static void set_load_weight(struct task_struct *p, bool update_load)
738{
739    int prio = p->static_prio - MAX_RT_PRIO;
740    struct load_weight *load = &p->se.load;
741
742    /*
743     * SCHED_IDLE tasks get minimal weight:
744     */
745    if (idle_policy(p->policy)) {
746        load->weight = scale_load(WEIGHT_IDLEPRIO);
747        load->inv_weight = WMULT_IDLEPRIO;
748        return;
749    }
750
751    /*
752     * SCHED_OTHER tasks have to update their load when changing their
753     * weight
754     */
755    if (update_load && p->sched_class == &fair_sched_class) {
756        reweight_task(p, prio);
757    } else {
758        load->weight = scale_load(sched_prio_to_weight[prio]);
759        load->inv_weight = sched_prio_to_wmult[prio];
760    }
761}

```

kernel/sched.c

#define WEIGHT_IDLEPRIO 2

#define WMULT_IDLEPRIO (1 << 31)



Core scheduler

- **Core scheduler:**
 - main scheduler: `schedule()`
 - Periodic Scheduler: `scheduler_tick()`

The main scheduler

- **schedule()**
 - 将**CPU**分配给与当前活动进程不同的另一个进程，会直接调用
 - 在一些检查点（比如从系统调用返回之后），内核会检查当前进程是否设置了重调度标志 **TIF_NEED_RESCHED**，如果被设置则调用
 - 如**scheduler_tick**会设置该标志

Cont.

- **__sched前缀**

- 可能调用**schedule**的函数，包括**schedule**自身

```
void __sched some_function(...) {  
    ...  
    schedule();  
    ...  
}
```

注：相关函数的代码编译之后，会放到目标文件的特定段**.sched.text**中。该信息使得内核在显示栈或相关信息时，**忽略所有与调度有关的调用**（调度器函数调用不是普通代码流的一部分）

Cont.

kernel/sched/core.c

```
3509asmlinkage __visible void __sched schedule(void)
3510{
3511    struct task_struct *tsk = current;
3512
3513    sched_submit_work(tsk); //处理块设备提交的任务
3514    do {
3515        preempt_disable();
3516        __schedule(false);
3517        sched_preempt_enable_no_resched();
3518    } while (need_resched());
3519}
3520EXPORT_SYMBOL(schedule);
```

```
178#define sched_preempt_enable_no_resched() \
179do { \
180    barrier(); \
181    preempt_count_dec(); \
182} while (0)
```

kernel/sched/core.c

```
3446     next = pick_next_task(rq, prev, &rf);
3447     clear_tsk_need_resched(prev);
3448     clear_preempt_need_resched();
3449
3450     if (likely(prev != next)) {
3451         rq->nr_switches++;
3452         rq->curr = next;
3453         .....
3467         ++*switch_count;
3468
3469         trace_sched_switch(preempt, prev, next);
3470
3471         /* Also unlocks the rq: */
3472         rq = context_switch(rq, prev, next, &rf);
3473     } else {
3474         .....
3476     }
3477
3478     balance_callback(rq);
3479 }
```

```
static void __sched __schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;
    .....
}
```


Context switch

kernel/sched/core.c

```
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
{
    struct mm_struct *mm, *oldmm;
    prepare_task_switch(rq, prev, next); //体系结构相关代码
    mm = next->mm;
    oldmm = prev->active_mm;
    ..
    if (!mm) {
        next->active_mm = oldmm;
        mmgrab(oldmm);
        enter_lazy_tlb(oldmm, next); //通知底层体系结构不需要切换虚拟地址空间的用户空间部分
    } else
        switch_mm_irqs_off(oldmm, mm, next);
}
```

switch_mm 更换通过 **task_struct->mm** 描述的内存管理上下文。该工作的细节取决于处理器，主要包括加载页表、刷出地址转换后备缓冲器（部分或全部）、向内存管理单元（**MMU**）提供新的信息

```
if (!prev->mm) {
    prev->active_mm = NULL; //重置 “借来” 的用户空间
    rq->prev_mm = oldmm;
}
...
/* Here we just switch the register state and the stack. */
switch_to(prev, next, prev);
barrier();

return finish_task_switch(prev); //清理工作，可以正确地释放锁
}
```

New task

- sched_fork

- 初始化新进程与调度相关的字段
- 建立数据结构
- 确定进程的动态优先级

kernel/sched/core.c

```
void sched_fork(unsigned long clone_flags, struct task_struct *p)
{
```

```
    unsigned long flags;
```

```
    __sched_fork(clone_flags, p);
```

```
    p->state = TASK_NEW;
```

```
    p->prio = current->normal_prio;
```

```
    if (unlikely(p->sched_reset_on_fork)) {
```

```
        if (!((task_has_dl_policy(p) || task_has_rt_policy(p))) {
```

```
            p->policy = SCHED_NORMAL;
```

```
            p->static_prio = NICE_TO_PRIO(0);
```

```
            p->rt_priority = 0;
```

```
        } else if (PRIO_TO_NICE(p->static_prio) < 0)
```

```
            .....}
```

```
        .....
        if (dl_prio(p->prio))    return -EAGAIN;
```

```
        else if (rt_prio(p->prio))    p->sched_class = &rt_sched_class;
```

```
        else    p->sched_class = &fair_sched_class;    .....
```

```
        if (p->sched_class->task_fork)    p->sched_class->task_fork(p);.....}
```

```
2144static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
2145{
2146    p->on_rq = 0;
2147
2148    p->se.on_rq = 0;
2149    p->se.exec_start = 0;
2150    p->se.sum_exec_runtime = 0;
2151    p->se.prev_sum_exec_runtime = 0;
2152    p->se.nr_migrations = 0;
2153    p->se.vruntime = 0;
2154    INIT_LIST_HEAD(&p->se.group_node);
2155
2156#ifdef CONFIG_FAIR_GROUP_SCHED
2157    p->se.cfs_rq = NULL;
2158#endif
2159
2160#ifdef CONFIG_SCHEDSTATS
2161    /* Even if schedstat is disabled, there should not be garbage */
2162    memset(&p->se.statistics, 0, sizeof(p->se.statistics));
2163#endif
2164
2165    RB_CLEAR_NODE(&p->dl.rb_node);
2166    init_dl_task_timer(&p->dl);
2167    init_dl_inactive_task_timer(&p->dl);
2168    __dl_clear_params(p);
2169
2170    INIT_LIST_HEAD(&p->rt.run_list);
2171    p->rt.timeout = 0;
2172    p->rt.time_slice = sched_rr_timeslice;
2173    p->rt.on_rq = 0;
2174    p->rt.on_list = 0;
2175
2176#ifdef CONFIG_PREEMPT_NOTIFIERS
2177    INIT_HLIST_HEAD(&p->preempt_notifiers);
2178#endif
2179
2180    init_numa_balancing(clone_flags, p);
2181}
```

Completely Fair Scheduling Class

kernel/sched/fair.c

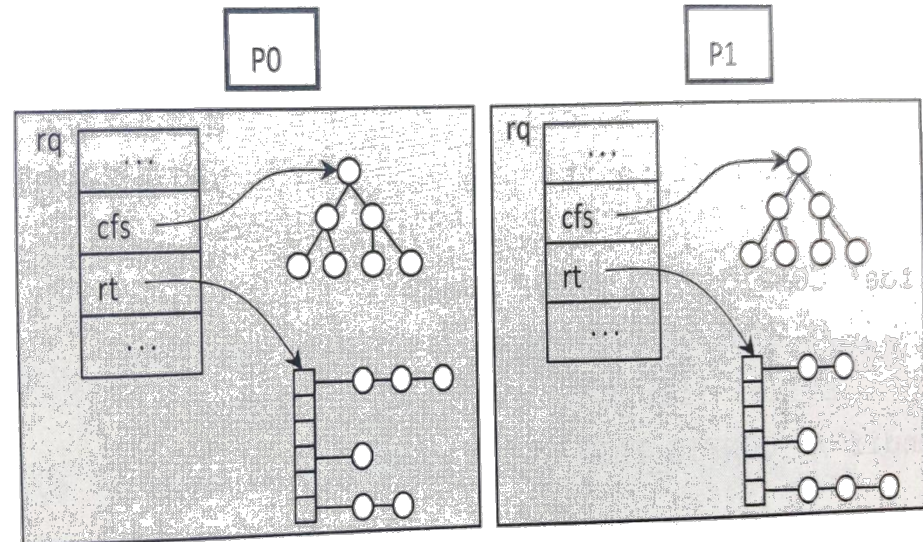
```
static const struct sched_class fair_sched_class = {  
    .next = &idle_sched_class,  
    .enqueue_task = enqueue_task_fair,  
    .dequeue_task = dequeue_task_fair,  
    .yield_task = yield_task_fair,  
    .check_preempt_curr = check_preempt_wakeup,  
    .pick_next_task = pick_next_task_fair,  
    .put_prev_task = put_prev_task_fair,  
    ...  
    .set_curr_task = set_curr_task_fair,  
    .task_tick = task_tick_fair,  
    .task_new = task_new_fair,  
};
```

cfs_rq

kernel/sched/sched.h

```
struct cfs_rq {  
    struct load_weight load;  
    unsigned long nr_running;  
    u64 min_vruntime;  
    .....  
    struct rb_root tasks_timeline;  
    struct rb_node *rb_leftmost;  
    struct sched_entity *curr;  
    .....  
}
```

○ 就绪任务



Cont.

- **nr_running**
 - 可运行进程的数目
- **Load**
 - 队列中进程的总负荷
- **min_vruntime**
 - 最小虚拟运行时间，是实现与就绪队列相关的虚拟时钟的基础。
 - **min_vruntime**可能比最左边的树结点的**vruntime**大
- **tasks_timeline**
 - 在按时间排序的红黑树中管理进程
- **rb_leftmost**
 - 指向树最左边的结点
- **curr**
 - 当前进程的调度实体

虚拟时钟

- 通过实际时钟和权重推算
- **update_curr**
 - 所有与虚拟时钟有关的计算
 - 在系统各个地方调用（包括周期性调度器）

Cont.

kernel/sched/fair.c

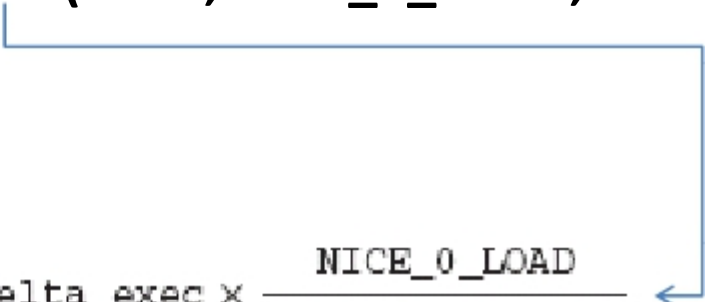
```
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_of(cfs_rq)->clock; —————→ rq_of: 确定与CFS就绪队
    unsigned long delta_exec;                               列相关的struct rq实例
    if (unlikely(!curr)) //就绪队列上没有进程在执行
        return;
    delta_exec = now - curr->exec_start;.....
    curr->exec_start = now;
    .....
    curr->sum_exec_runtime += delta_exec;
    .....
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);
    .....
    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```


Cont.

kernel/sched/fair.c

```
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);
    return delta;
}
```

$\text{delta_exec_weighted} = \text{delta_exec} \times \frac{\text{NICE_0_LOAD}}{\text{curr->load.weight}}$



```

496static void update_min_vruntime(struct cfs_rq *cfs_rq)
497{
498    struct sched_entity *curr = cfs_rq->curr;
499    struct rb_node *leftmost = rb_first_cached(&cfs_rq->tasks_timeline);
500    u64 vruntime = cfs_rq->min_vruntime;
501    if (curr) {
502        if (curr->on_rq)
503            vruntime = curr->vruntime;
504        else
505            curr = NULL;
506    }
507    if (leftmost) { /* non-empty tree */
508        struct sched_entity *se;
509        se = rb_entry(leftmost, struct sched_entity, run_node);
510        if (!curr)
511            vruntime = se->vruntime;
512        else
513            vruntime = min_vruntime(vruntime, se->vruntime); //取最左孩子和
                                                                //curr最小值
514    }
515    /* ensure we never gain time by being placed backwards. */
516    cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime, vruntime); //单增
517}
518
519.....}

```

Queue Manipulation

输入参数: rq/task_struct/wakeup

```
5081 enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
5082 {
5083     struct cfs_rq *cfs_rq;
5084     struct sched_entity *se = &p->se;
5085     .....
5102     for_each_sched_entity(se) {
5103         if (se->on_rq)
5104             break;
5105         cfs_rq = cfs_rq_of(se);
5106         enqueue_entity(cfs_rq, se, flags);
5107     }
5108     .....
5119 }
5120 .....
3859 enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
3860 {
3861     bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
3862     bool curr = cfs_rq->curr == se;
3863     if (renorm && curr)
3864         se->vruntime += cfs_rq->min_vruntime;
3865     update_curr(cfs_rq);
3866     if (renorm && !curr)
3867         se->vruntime += cfs_rq->min_vruntime;
3868     update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);
3869     update_cfs_group(se);
3870     enqueue_runnable_load_avg(cfs_rq, se);
3871     account_entity_enqueue(cfs_rq, se);
3872     if (flags & ENQUEUE_WAKEUP)
3873         place_entity(cfs_rq, se, 0); //0刚唤醒1新建任务, 调整vruntime
3874     .....
3901     if (!curr)
3902         __enqueue_entity(cfs_rq, se); //根据vruntime插入红黑树
3903     se->on_rq = 1;
3904     .....
3909 }
```

Latency Tracking

- **sysctl_sched_latency**
 - 延迟周期，在此期间每个就绪进程至少运行一次
 - `/proc/sys/kernel/sched_latency_ns`
 - 默认6ms
- **sched_nr_latency**
 - 延迟周期内处理的最大活动进程数目
 - 如果活动进程数超出该限制，则延迟周期按比例线性扩展
 - `sysctl_sched_latency/sysctl_sched_min_granularity`
 - `sysctl_sched_min_granularity`由
`/proc/sys/kernel/sched_min_granularity_ns`设置， 默认0.75ms

Cont.

- **__sched_period**
 - 确定延迟周期的长度
 - 通常为sysctl_sched_latency
 - 如果运行进程多，有可能按比例线性扩展

$$\text{sysctl_sched_latency} \times \frac{\text{nr_running}}{\text{sched_nr_latency}}$$

```
644static u64 __sched_period(unsigned long nr_running)
645{
646    if (unlikely(nr_running > sched_nr_latency))
647        return nr_running * sysctl_sched_min_granularity;
648    else
649        return sysctl_sched_latency;
650}
```

Cont.

时间片分配

kernel/sched/fair.c

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);

    for_each_sched_entity(se) {
        struct load_weight *load;
        struct load_weight lw;

        cfs_rq = cfs_rq_of(se);
        load = &cfs_rq->load;

        if (unlikely(!se->on_rq)) {
            lw = cfs_rq->load;

            update_load_add(&lw, se->load.weight);
            load = &lw;
        }
        slice = __calc_delta(slice, se->load.weight, load);
    }
    return slice;
}
```

$$\text{slice} = \text{slice} * \frac{\text{se} \rightarrow \text{load.weight}}{\text{load}}$$

The Periodic Scheduler

- **scheduler_tick**
 - 更新调度统计信息
 - 周期性调度算法

```
9774static void task_tick_fair
(struct rq *rq, struct task_struct *curr, int queued)
9775{
9776    struct cfs_rq *cfs_rq;
9777    struct sched_entity *se = &curr->se;
9778
9779    for_each_sched_entity(se) {
9780        cfs_rq = cfs_rq_of(se);
9781        entity_tick(cfs_rq, se, queued);
9782    }
```

```
.....
9788}

4170entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
4171{
4175    update_curr(cfs_rq);
4180    update_load_avg(cfs_rq, curr, UPDATE_TG);
4181    update_cfs_group(curr);
.....
4200    if (cfs_rq->nr_running > 1)
4201        check_preempt_tick(cfs_rq, curr);
4202}
```

kernel/sched/core.c

```
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    ...
    update_rq_clock(rq)
    curr->sched_class->task_tick(rq, curr, 0);
    cpu_load_update_active(rq);
    ... }
```

```

4014 check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
4015 {
4016     unsigned long ideal_runtime, delta_exec;
4017     struct sched_entity *se;
4018     s64 delta;
4019
4020     ideal_runtime = sched_slice(cfs_rq, curr); //时间延迟计算出来的理想运行时间
4021     delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime; //实际运行时间
4022     if (delta_exec > ideal_runtime) {
4023         resched_curr(rq_of(cfs_rq));
4024         /*
4025          * The current task ran long enough, ensure it doesn't get
4026          * re-elected due to buddy favours.
4027          */
4028         clear_buddies(cfs_rq, curr);
4029         return;
4030     }
4037     if (delta_exec < sysctl_sched_min_granularity)
4038         return;
4040     se = __pick_first_entity(cfs_rq);
4041     delta = curr->vruntime - se->vruntime;
4043     if (delta < 0)
4044         return;
4046     if (delta > ideal_runtime)
4047         resched_curr(rq_of(cfs_rq));
4048 }
4049

```


pick_next_task_fair

```
6637static struct task_struct *
6638pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
6639{
6640    struct cfs_rq *cfs_rq = &rq->cfs;
6641    struct sched_entity *se;
6642    struct task_struct *p;
6643    int new_tasks;
6644
6645again:
6646    if (!cfs_rq->nr_running)
6647        goto idle;
6648
6649    .....
6661    do {
6662        struct sched_entity *curr = cfs_rq->curr;
6663
6664        .....
6670        if (curr) {
6671            if (curr->on_rq)
6672                update_curr(cfs_rq);
6673            else
6674                curr = NULL;
6675
6676        .....
6692        se = pick_next_entity(cfs_rq, curr);
6693        cfs_rq = group_cfs_rq(se);
6694    } while (cfs_rq);
6695
6696    .....
6771}
6772
```

```
4092static struct sched_entity *
4093pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
4094{
4095    struct sched_entity *left = __pick_first_entity(cfs_rq);
4096    struct sched_entity *se;
4102    if (!left || (curr && entity_before(curr, left)))
4103        left = curr;
4105    se = left; /* ideally we run the leftmost entity */
4111    if (cfs_rq->skip == se) {
4112        struct sched_entity *second;
4113
4114        if (se == curr) {
4115            second = __pick_first_entity(cfs_rq);
4116        } else {
4117            second = __pick_next_entity(se);
4118            if (!second || (curr && entity_before(curr, second)))
4119                second = curr;
4120        }
4121
4122        if (second && wakeup_preempt_entity(second, left) < 1)
4123            se = second;
4124    }
4129    if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
4130        se = cfs_rq->last;
4135    if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
4136        se = cfs_rq->next;
4138    clear_buddies(cfs_rq, se);
4140    return se;
4141}
```

The Real-Time Scheduling Class

- **SCHED_RR**

- 时间片，在进程运行时减少，到期后重置为初始值，进程置于队列的末尾

- **SCHED_FIFO**

- 无时间片
- 被调度执行后可以运行任意长时间

- 在编写实时应用程序时，应小心

- 无限循环，循环体内不进入睡眠，则系统无法运行。

Cont.

kernel/sched/rt.c

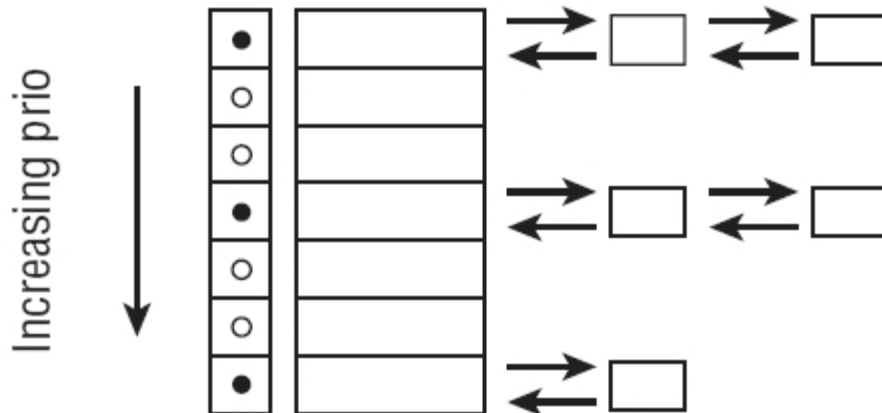
```
const struct sched_class rt_sched_class = {  
    .next = &fair_sched_class,  
    .enqueue_task = enqueue_task_rt,  
    .dequeue_task = dequeue_task_rt,  
    .yield_task = yield_task_rt,  
    .check_preempt_curr = check_preempt_curr_rt,  
    .pick_next_task = pick_next_task_rt,  
    .put_prev_task = put_prev_task_rt,  
    .set_curr_task = set_curr_task_rt,  
    .task_tick = task_tick_rt,  
};
```

```
kernel/sched/sched.h  
struct rq {  
    ...  
    rt_rq rt;  
    ...  
}
```

Cont.

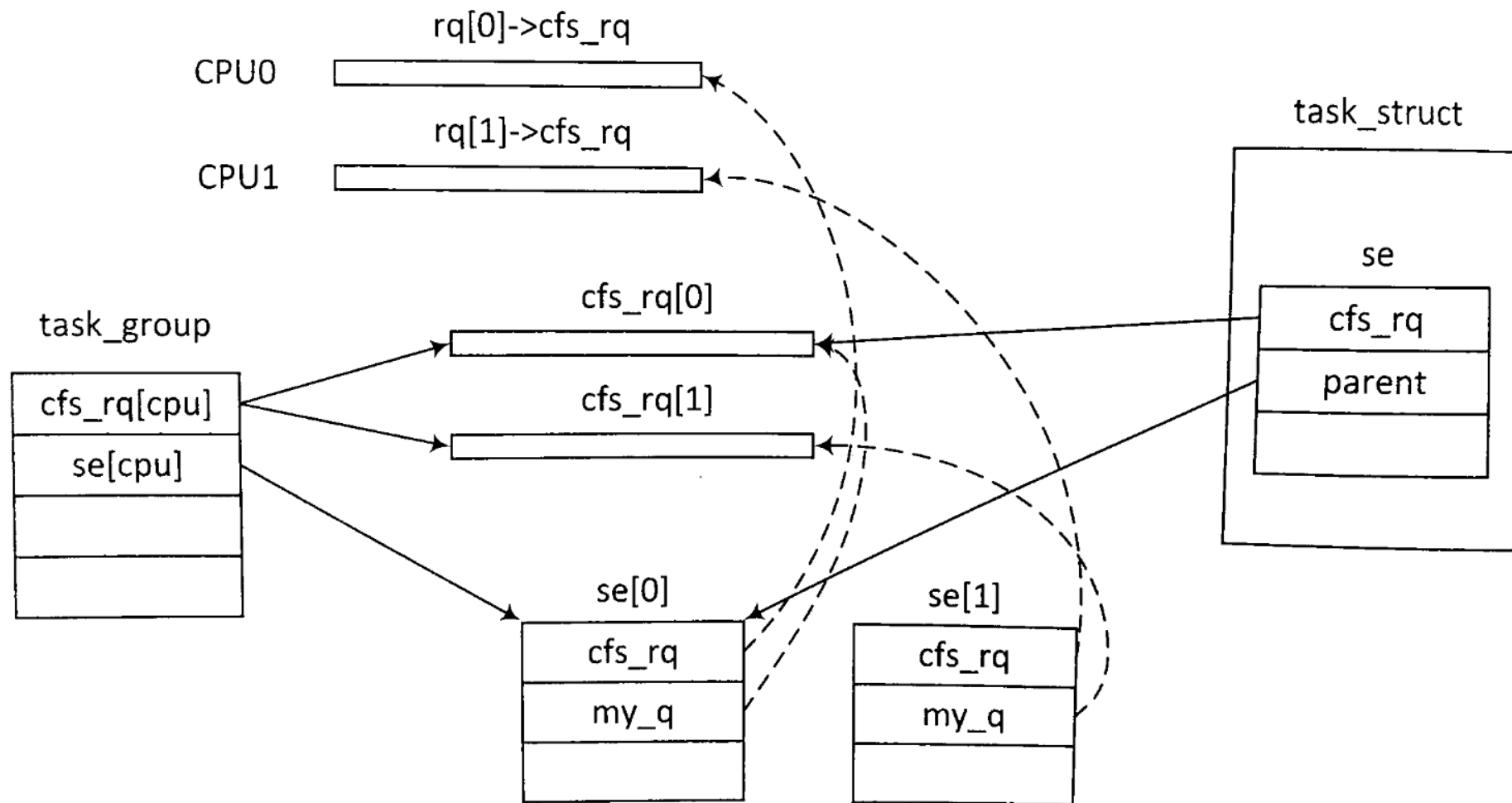
kernel/sched/sched.h

```
struct rt_prio_array {  
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit  
    for delimiter */  
    struct list_head queue[MAX_RT_PRIO];  
};  
struct rt_rq {  
    struct rt_prio_array active;  
};
```



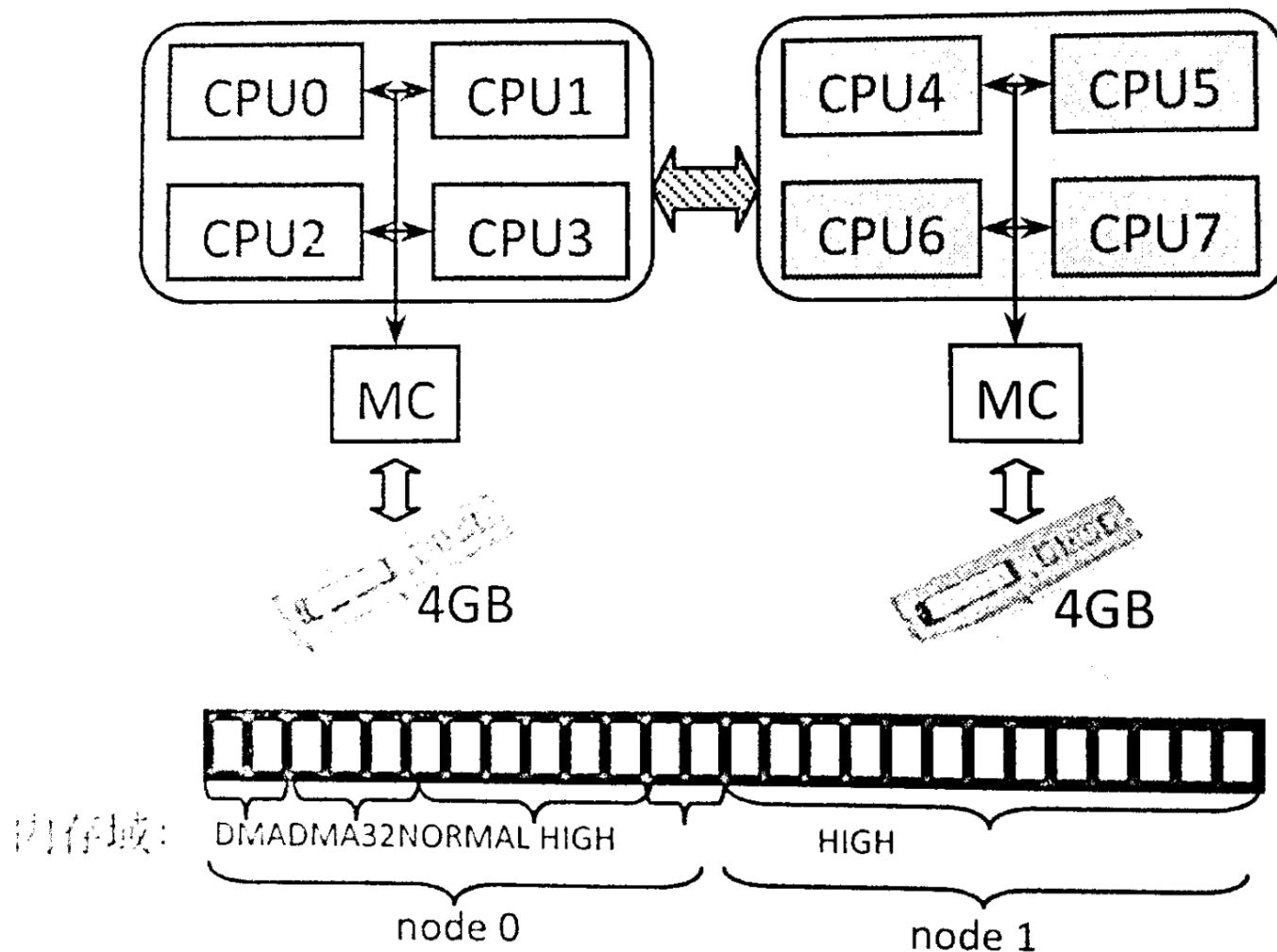
`prio_array_t`的结构如下:

```
struct prio_array {  
    int nr_active; /*本进程组中进程个数*/  
    struct list_head queue[MAX_PRIO];  
    /*每个优先级的进程队列*/  
    unsigned long bitmap[BITMAP_SIZE];  
    /*上述进程队列的索引位图*/  
};
```



Load_balance

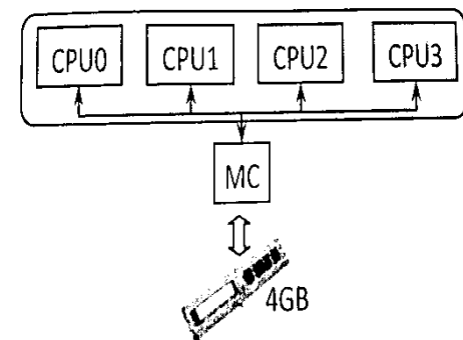
NUMA 架构



sched_domain level:

1. HT (Hyper-threading) 或 Simultaneous MultiThreading (SMT)
2. Core
3. MC
4. NUMA

UMA 架构



作业3

- 分别用**fork**、**vfork**创建进程，设计实验查看二者的区别（资源的分配和共享、执行顺序）
- 显示进程的相关信息
（**pid,ppid,state,parent,real_parent,....**）并捡取重要的**10+**个加以解释
- 拓展阅读：
 - 《奔跑吧》
 - 《Linux技术内幕》进程调度相关章节