一、进程原理

1、进程

Linux内核把进程称为任务(task)，进程的虚拟地址空间分为用户虚拟地址空间和内核虚拟地址空间，所有进程共享内核虚拟地址空间，每个进程有独立的用户空间虚拟地址空间。

进程有两种特殊形式：没有用户虚拟地址空间的进程称为内核线程，共享用户虚拟地址空间的进程称为用户线程。通用在不会引起混淆的情况下把用户线程简称为线程。共享同一个用户虚拟地址空间的所有用户线程组成一个线程组。

C标准库的进程专业术语和Linux内核的进程专业术语对应关系如下：

C标准库的进程专业术语	Linux内核的
包含多个线程的进程	线程组
只有一个线程的进程	进程或任务
线程	共享用户虚拟地址空间的i

2、Linux进程四要素：

a.有一段程序供其执行。

b.有进程专用的系统堆栈空间。

c.在内核有task_struct数据结构；

d.有独立的存储空间，拥有专有的用户空间。

如果只具备前三条而缺少第四条，则称为“线程”。如果完全没有用户空间，就称为“内核线程”；而如果共享用户空间映射就称为“用户线程”。内核为每个进程分配一个task_struct结构时。实际分配两个连续物理页面(8192字节)，数据结构task_struct的大小约占1kb字节左右，进程的系统空间堆栈的大小约为7kb字节（不能扩展，是静态确定的）。

3、进程描述符task_struct数据结构内核源码，其主要核心成员如下：

include > linux > C sched.h > task_struct

```c

struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state;
	void				*stack;
	atomic_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;

```

```c
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state;
	void				*stack;
	atomic_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;
#ifdef CONFIG_SMP
	struct llist_node		wake_entry;
```

```c
    int                 on_cpu;
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /* Current CPU: */
    unsigned int            cpu;
#endif
    unsigned int            wakee_flips;
    unsigned long           wakee_flip_decay_ts;
    struct task_struct      *last_wakee;
    int                 wake_cpu;
#endif
    int                 on_rq;
    int                 prio;
    int                 static_prio;
    int                 normal_prio;
    unsigned int            rt_priority;
    const struct sched_class    *sched_class;
    struct sched_entity     se;
    struct sched_rt_entity      rt;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group       *sched_task_group;
#endif
    struct sched_dl_entity      dl;
#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* List of struct preempt_notifier: */
    struct hlist_head       preempt_notifiers;
#endif
#ifdef CONFIG_BLK_DEV_IO_TRACE
```

```
    unsigned int            btrace_seq;
#endif
    unsigned int            policy;
    int             nr_cpus_allowed;
    cpumask_t           cpus_allowed;
#ifdef CONFIG_PREEMPT_RCU
    int             rcu_read_lock_nesting;
    union rcu_special       rcu_read_unlock_special;
    struct list_head        rcu_node_entry;
    struct rcu_node         *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
    unsigned long           rcu_tasks_nvcsw;
    bool            rcu_tasks_holdout;
    struct list_head        rcu_tasks_holdout_list;
    int             rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */
    struct sched_info       sched_info;
    struct list_head        tasks;
```

// 配置SMP（SMP（对称多处理器）系统的应用越来越广泛，规模也越来越大，但由于传统的SMP系统中，所有处理器都共享系统总线，因此当处理器的数目增大时，系统总线的竞争冲突加大，系统总线将成为瓶颈，所以目前SMP系统的CPU数目一般只有数十个，可扩展能力受到极大限制。NUMA技术有效结合了SMP系统易编程性和MPP（大规模并行）系统易扩展性的特点，较好解决了SMP系统的可扩展性问题，已成为当今高性能服务器的主流体系结构之一。

```c
#ifdef CONFIG_SMP
    struct plist_node       pushable_tasks;
    struct rb_node          pushable_dl_tasks;
#endif
    struct mm_struct        *mm;
    struct mm_struct        *active_mm;
    /* Per-thread vma caching: */
    struct vmacache         vmacache;
#ifdef SPLIT_RSS_COUNTING
    struct task_rss_stat    rss_stat;
#endif
    int         exit_state;
    int         exit_code;
    int         exit_signal;
    /* The signal sent when the parent dies: */
    int         pdeath_signal;
    /* JOBCTL_*, siglock protected: */
    unsigned long           jobctl;
    /* Used for emulating ABI behavior of previous Linux versions: */
    unsigned int            personality;
    /* Scheduler bits, serialized by scheduler locks: */
    unsigned            sched_reset_on_fork:1;
    unsigned            sched_contributes_to_load:1;
    unsigned            sched_migrated:1;
    unsigned            sched_remote_wakeup:1;
    /* Force alignment to the next boundary: */
    unsigned            :0;
```

```
	/* Unserialized, strictly 'current' */
	/* Bit to tell LSMs we're in execve(): */
	unsigned			in_execve:1;
	unsigned			in_iowait:1;
#ifndef TIF_RESTORE_SIGMASK
	unsigned			restore_sigmask:1;
#endif
#ifdef CONFIG_MEMCG
	unsigned			memcg_may_oom:1;
#ifndef CONFIG_SLOB
	unsigned			memcg_kmem_skip_account:1;
#endif
#endif
#ifdef CONFIG_COMPAT_BRK
	unsigned			brk_randomized:1;
#endif
#ifdef CONFIG_CGROUPS
	/* disallow userland-initiated cgroup migration */
	unsigned			no_cgroup_migration:1;
#endif
	unsigned long			atomic_flags; /* Flags requiring atomic acces
s. */
	struct restart_block		restart_block;
	pid_t				pid;
	pid_t				tgid;
#ifdef CONFIG_CC_STACKPROTECTOR
	/* Canary value for the -fstack-protector GCC feature: */
```

```
    unsigned long           stack_canary;
#endif
    /*
     * Pointers to the (original) parent process, youngest child, young
er sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->real_parent->pid)
     */
    /* Real parent process: */
    struct task_struct __rcu    *real_parent;
    /* Recipient of SIGCHLD, wait4() reports: */
    struct task_struct __rcu    *parent;
    /*
     * Children/sibling form the list of natural children:
     */
    struct list_head        children;
    struct list_head        sibling;
    struct task_struct      *group_leader;
    /*
     * 'ptraced' is the list of tasks this task is using ptrace() on.
     *
     * This includes both natural children and PTRACE_ATTACH target
s.
     * 'ptrace_entry' is this task's link on the p->parent->ptraced list.
     */
    struct list_head        ptraced;
    struct list_head        ptrace_entry;
```

```c
	/* PID/PID hash table linkage. */
	struct pid_link			pids[PIDTYPE_MAX];
	struct list_head		thread_group;
	struct list_head		thread_node;
	struct completion		*vfork_done;
	/* CLONE_CHILD_SETTID: */
	int __user			*set_child_tid;
	/* CLONE_CHILD_CLEARTID: */
	int __user			*clear_child_tid;
	u64				utime;
	u64				stime;
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
	u64				utimescaled;
	u64				stimescaled;
#endif
	u64				gtime;
	struct prev_cputime		prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
	seqcount_t			vtime_seqcount;
	unsigned long long		vtime_snap;
	enum {
		/* Task is sleeping or running in a CPU with VTIME inactive: */
		VTIME_INACTIVE = 0,
		/* Task runs in userspace in a CPU with VTIME active: */
		VTIME_USER,
		/* Task runs in kernelspace in a CPU with VTIME active: */
		VTIME_SYS,
```

```
	} vtime_snap_whence;
#endif
#ifdef CONFIG_NO_HZ_FULL
	atomic_t			tick_dep_mask;
#endif
	/* Context switch counts: */
	unsigned long			nvcsw;
	unsigned long			nivcsw;
	/* Monotonic time in nsecs: */
	u64				start_time;
	/* Boot based time in nsecs: */
	u64				real_start_time;
	/* MM fault and swap info: this can arguably be seen as either mm-specific or thread-specific: */
	unsigned long			min_flt;
	unsigned long			maj_flt;
#ifdef CONFIG_POSIX_TIMERS
	struct task_cputime		cputime_expires;
	struct list_head		cpu_timers[3];
#endif
	/* Process credentials: */
	/* Tracer's credentials at attach: */
	const struct cred __rcu		*ptracer_cred;
	/* Objective and real subjective task credentials (COW): */
	const struct cred __rcu		*real_cred;
	/* Effective (overridable) subjective task credentials (COW): */
	const struct cred __rcu		*cred;
```

```
	/*
	 * executable name, excluding path.
	 *
	 * - normally initialized setup_new_exec()
	 * - access it with [gs]et_task_comm()
	 * - lock it with task_lock()
	 */
	char				comm[TASK_COMM_LEN];
	struct nameidata		*nameidata;
#ifdef CONFIG_SYSVIPC
	struct sysv_sem			sysvsem;
	struct sysv_shm			sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
	unsigned long			last_switch_count;
#endif
	/* Filesystem information: */
	struct fs_struct		*fs;
	/* Open file information: */
	struct files_struct		*files;
	/* Namespaces: */
	struct nsproxy			*nsproxy;
	/* Signal handlers: */
	struct signal_struct		*signal;
	struct sighand_struct		*sighand;
	sigset_t			blocked;
	sigset_t			real_blocked;
```

```
	/* Restored if set_restore_sigmask() was used: */
	sigset_t			saved_sigmask;
	struct sigpending		pending;
	unsigned long			sas_ss_sp;
	size_t				sas_ss_size;
	unsigned int			sas_ss_flags;
	struct callback_head		*task_works;
	struct audit_context		*audit_context;
#ifdef CONFIG_AUDITSYSCALL
	kuid_t				loginuid;
	unsigned int			sessionid;
#endif
	struct seccomp			seccomp;
	/* Thread group tracking: */
	u32				parent_exec_id;
	u32				self_exec_id;
	/* Protection against (de-)allocation: mm, files, fs, tty, keyrings, m
ems_allowed, mempolicy: */
	spinlock_t			alloc_lock;
	/* Protection of the PI data structures: */
	raw_spinlock_t			pi_lock;
	struct wake_q_node		wake_q;
#ifdef CONFIG_RT_MUTEXES
	/* PI waiters blocked on a rt_mutex held by this task: */
	struct rb_root			pi_waiters;
	struct rb_node			*pi_waiters_leftmost;
	/* Updated under owner's pi_lock and rq lock */
```

```
    struct task_struct      *pi_top_task;
    /* Deadlock detection and priority inheritance handling: */
    struct rt_mutex_waiter      *pi_blocked_on;
#endif
#ifdef CONFIG_DEBUG_MUTEXES
    /* Mutex deadlock detection: */
    struct mutex_waiter     *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
    unsigned int            irq_events;
    unsigned long           hardirq_enable_ip;
    unsigned long           hardirq_disable_ip;
    unsigned int            hardirq_enable_event;
    unsigned int            hardirq_disable_event;
    int             hardirqs_enabled;
    int             hardirq_context;
    unsigned long           softirq_disable_ip;
    unsigned long           softirq_enable_ip;
    unsigned int            softirq_disable_event;
    unsigned int            softirq_enable_event;
    int             softirqs_enabled;
    int             softirq_context;
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH         48UL
    u64             curr_chain_key;
    int             lockdep_depth;
```

```
	unsigned int			lockdep_recursion;
	struct held_lock		held_locks[MAX_LOCK_DEPTH];
	gfp_t				lockdep_reclaim_gfp;
#endif
#ifdef CONFIG_UBSAN
	unsigned int			in_ubsan;
#endif
	/* Journalling filesystem info: */
	void				*journal_info;
	/* Stacked block device info: */
	struct bio_list			*bio_list;
#ifdef CONFIG_BLOCK
	/* Stack plugging: */
	struct blk_plug			*plug;
#endif
	/* VM state: */
	struct reclaim_state		*reclaim_state;
	struct backing_dev_info		*backing_dev_info;
	struct io_context		*io_context;
	/* Ptrace state: */
	unsigned long			ptrace_message;
	siginfo_t			*last_siginfo;
	struct task_io_accounting	ioac;
#ifdef CONFIG_TASK_XACCT
	/* Accumulated RSS usage: */
	u64				acct_rss_mem1;
	/* Accumulated virtual memory usage: */
```

```c
    u64             acct_vm_mem1;
    /* stime + utime since last update: */
    u64             acct_timexpd;
#endif
#ifdef CONFIG_CPUSETS
    /* Protected by ->alloc_lock: */
    nodemask_t          mems_allowed;
    /* Seqence number to catch updates: */
    seqcount_t          mems_allowed_seq;
    int             cpuset_mem_spread_rotor;
    int             cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
    /* Control Group info protected by css_set_lock: */
    struct css_set __rcu        *cgroups;
    /* cg_list protected by css_set_lock and tsk->alloc_lock: */
    struct list_head        cg_list;
#endif
#ifdef CONFIG_INTEL_RDT_A
    int             closid;
#endif
#ifdef CONFIG_FUTEX
    struct robust_list_head __user  *robust_list;
#ifdef CONFIG_COMPAT
    struct compat_robust_list_head __user *compat_robust_list;
#endif
    struct list_head        pi_state_list;
```

```
	struct futex_pi_state		*pi_state_cache;
#endif
#ifdef CONFIG_PERF_EVENTS
	struct perf_event_context	*perf_event_ctxp[perf_nr_task_context
s];
	struct mutex			perf_event_mutex;
	struct list_head		perf_event_list;
#endif
#ifdef CONFIG_DEBUG_PREEMPT
	unsigned long			preempt_disable_ip;
#endif
#ifdef CONFIG_NUMA
	/* Protected by alloc_lock: */
	struct mempolicy		*mempolicy;
	short				il_next;
	short				pref_node_fork;
#endif
#ifdef CONFIG_NUMA_BALANCING
	int				numa_scan_seq;
	unsigned int			numa_scan_period;
	unsigned int			numa_scan_period_max;
	int				numa_preferred_nid;
	unsigned long			numa_migrate_retry;
	/* Migration stamp: */
	u64				node_stamp;
	u64				last_task_numa_placement;
	u64				last_sum_exec_runtime;
```

```
	struct callback_head		numa_work;
	struct list_head		numa_entry;
	struct numa_group		*numa_group;
	/*
	 * numa_faults is an array split into four regions:
	 * faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
	 * in this precise order.
	 *
	 * faults_memory: Exponential decaying average of faults on a per-node
	 * basis. Scheduling placement decisions are made based on these
	 * counts. The values remain static for the duration of a PTE scan.
	 * faults_cpu: Track the nodes the process was running on when a NUMA
	 * hinting fault was incurred.
	 * faults_memory_buffer and faults_cpu_buffer: Record faults per node
	 * during the current scan window. When the scan completes, the counts
	 * in faults_memory and faults_cpu decay and these values are copied.
	 */
	unsigned long			*numa_faults;
	unsigned long			total_numa_faults;
	/*
```

```
	 * numa_faults_locality tracks if faults recorded during the last
	 * scan window were remote/local or failed to migrate. The task s
can
	 * period is adapted based on the locality of the faults with differ
ent
	 * weights depending on whether they were shared or private fau
lts
	 */
	unsigned long			numa_faults_locality[3];
	unsigned long			numa_pages_migrated;
#endif /* CONFIG_NUMA_BALANCING */
	struct tlbflush_unmap_batch	tlb_ubc;
	struct rcu_head			rcu;
	/* Cache last used pipe for splice(): */
	struct pipe_inode_info		*splice_pipe;
	struct page_frag		task_frag;
#ifdef CONFIG_TASK_DELAY_ACCT
	struct task_delay_info		*delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
	int				make_it_fail;
#endif
	/*
	 * When (nr_dirtied >= nr_dirtied_pause), it's time to call
	 * balance_dirty_pages() for a dirty throttling pause:
	 */
	int				nr_dirtied;
```

```c
	int				nr_dirtied_pause;
	/* Start of a write-and-pause period: */
	unsigned long			dirty_paused_when;
#ifdef CONFIG_LATENCYTOP
	int				latency_record_count;
	struct latency_record		latency_record[LT_SAVECOUNT];
#endif
	/*
	 * Time slack values; these are used to round up poll() and
	 * select() etc timeout values. These are in nanoseconds.
	 */
	u64				timer_slack_ns;
	u64				default_timer_slack_ns;
#ifdef CONFIG_KASAN
	unsigned int			kasan_depth;
#endif
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
	/* Index of current stored address in ret_stack: */
	int				curr_ret_stack;
	/* Stack of return addresses for return function tracing: */
	struct ftrace_ret_stack		*ret_stack;
	/* Timestamp for last schedule: */
	unsigned long long		ftrace_timestamp;
	/*
	 * Number of functions that haven't been traced
	 * because of depth overrun:
	 */
```

```
	atomic_t			trace_overrun;
	/* Pause tracing: */
	atomic_t			tracing_graph_pause;
#endif
#ifdef CONFIG_TRACING
	/* State flags for use by tracers: */
	unsigned long			trace;
	/* Bitmask and counter of trace recursion: */
	unsigned long			trace_recursion;
#endif /* CONFIG_TRACING */
#ifdef CONFIG_KCOV
	/* Coverage collection mode enabled for this task (0 if disabled):
*/
	enum kcov_mode			kcov_mode;
	/* Size of the kcov_area: */
	unsigned int			kcov_size;
	/* Buffer for coverage collection: */
	void				*kcov_area;
	/* KCOV descriptor wired with this task or NULL: */
	struct kcov			*kcov;
#endif
#ifdef CONFIG_MEMCG
	struct mem_cgroup		*memcg_in_oom;
	gfp_t				memcg_oom_gfp_mask;
	int				memcg_oom_order;
	/* Number of pages to reclaim on returning to userland: */
	unsigned int			memcg_nr_pages_over_high;
```

```
#endif
#ifdef CONFIG_UPROBES
	struct uprobe_task		*utask;
#endif
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE
)
	unsigned int			sequential_io;
	unsigned int			sequential_io_avg;
#endif
#ifdef CONFIG_DEBUG_ATOMIC_SLEEP
	unsigned long			task_state_change;
#endif
	int				pagefault_disabled;
#ifdef CONFIG_MMU
	struct task_struct		*oom_reaper_list;
#endif
#ifdef CONFIG_VMAP_STACK
	struct vm_struct		*stack_vm_area;
#endif
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/* A live task holds one reference: */
	atomic_t			stack_refcount;
#endif
#ifdef CONFIG_LIVEPATCH
	int patch_state;
#endif
#ifdef CONFIG_SECURITY
```

```
    /* Used by LSM modules for access restriction: */
    void                *security;
#endif
    /* CPU-specific state of this task: */
    struct thread_struct        thread;
    /*
     * WARNING: on x86, 'thread_struct' contains a variable-sized
     * structure.  It *MUST* be at the end of 'task_struct'.
     *
     * Do not put anything below here!
     */
};
```

4、创建新进程

在Linux内核中，新进程是从一个已经存在的进程复制出来的，内核使用静态数据结构造出0号内核线程，0号内核线程分叉生成1号内核线程和2号内核线程（kthreadd线程）。1号内核线程完成初始化以后装载用户程序，变成1号进程，其他进程都是1号进程或者它的子孙进程分叉生成的；其他内核线程是kthreadd线程分叉生成的。

3个系统调用可以用来创建新的进程：

a.fork(分叉)：子进程是父进程的一个副本，采用定时复制技术。

b.vfor：用于创建子进程，之后子进程立即调用execve以装载新程序的情况，为了避免复制物理页，父进程会睡眠等待子进程装载新程序。现在fork采用了定时复制技术，vfork失去了速度优势，已经被废弃。

c.clone（克隆）：可以精确地控制子进程和父进程共享哪些资源。这个系统调用的主要用处是可供pthread库用来创建线程。clone是功能最

齐全的函数，参数多使用复杂，fork是clone的简化函数。

Linux内核定义系统调用的独特方式，目前以系统调用fork为例：

系统调用的函数名称以"sys_"开头，创建新进程的3个系统调用在文件"kernel/fork.c"中，它们把工作委托给函数_do_fork。

kernel > C fork.c > ...

```c

long _do_fork(unsigned long clone_flags,
        unsigned long stack_start,
        unsigned long stack_size,
        int __user *parent_tidptr,
        int __user *child_tidptr,
        unsigned long tls)
{
    struct task_struct *p;
    int trace = 0;
    long nr;

```

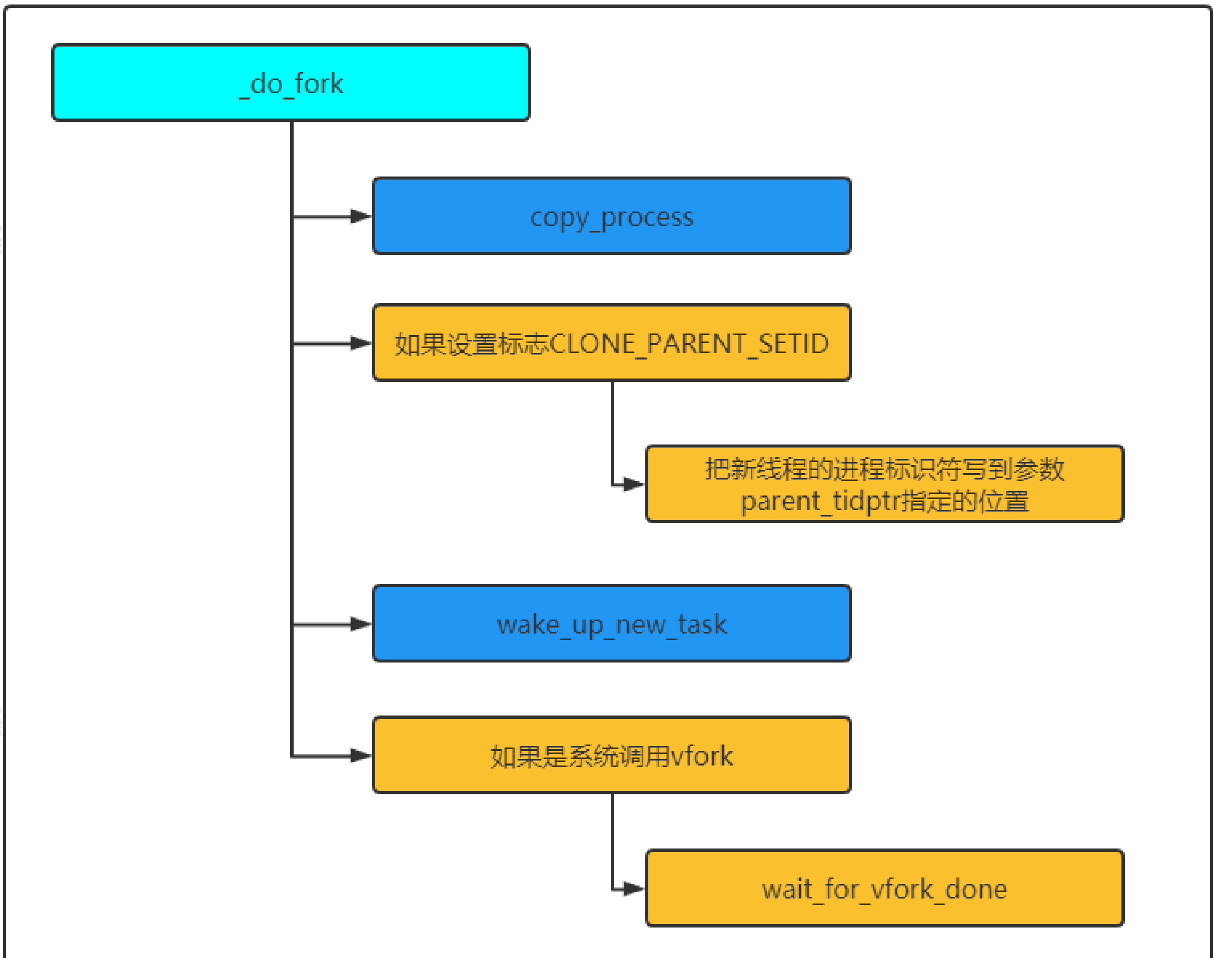
函数_do_fork()内核源码如下：

```c
long _do_fork(unsigned long clone_flags,
        unsigned long stack_start,
        unsigned long stack_size,
        int __user *parent_tidptr,
        int __user *child_tidptr,
        unsigned long tls)
{
    struct task_struct *p;
    int trace = 0;
    long nr;
    /*
     * Determine whether and which event to report to ptracer.  When
     * called from kernel_thread or CLONE_UNTRACED is explicitly
     * requested, no event is reported; otherwise, report if the event
```

```
	 * for the type of forking is enabled.
	 */
	if (!(clone_flags & CLONE_UNTRACED)) {
		if (clone_flags & CLONE_VFORK)
			trace = PTRACE_EVENT_VFORK;
		else if ((clone_flags & CSIGNAL) != SIGCHLD)
			trace = PTRACE_EVENT_CLONE;
		else
			trace = PTRACE_EVENT_FORK;
		if (likely(!ptrace_event_enabled(current, trace)))
			trace = 0;
	}
	p = copy_process(clone_flags, stack_start, stack_size,
			 child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
	add_latent_entropy();
	/*
	 * Do this prior waking up the new thread - the thread pointer
	 * might get invalid after that point, if the thread exits quickly.
	 */
	if (!IS_ERR(p)) {
		struct completion vfork;
		struct pid *pid;
		trace_sched_process_fork(current, p);
		pid = get_task_pid(p, PIDTYPE_PID);
		nr = pid_vnr(pid);
		if (clone_flags & CLONE_PARENT_SETTID)
			put_user(nr, parent_tidptr);
```

```
        if (clone_flags & CLONE_VFORK) {
            p->vfork_done = &vfork;
            init_completion(&vfork);
            get_task_struct(p);
        }
        wake_up_new_task(p);
        /* forking complete and child started to run, tell ptracer */
        if (unlikely(trace))
            ptrace_event_pid(trace, pid);
        if (clone_flags & CLONE_VFORK) {
            if (!wait_for_vfork_done(p, &vfork))
                ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
        }
        put_pid(pid);
    } else {
        nr = PTR_ERR(p);
    }
    return nr;
}
```

Linux内核函数_do_fork()执行流程如下图所示：



函数_do_fork()执行流程

kernel > C fork.c > copy_process(unsigned long, unsigned long, unsigned long, int __user *, pid *, int, unsigned long, int)

```c
static __latent_entropy struct task_struct *copy_process(
					unsigned long clone_flags,
					unsigned long stack_start,
					unsigned long stack_size,
					int __user *child_tidptr,
					struct pid *pid,
					int trace,
					unsigned long tls,
					int node)
{
	int retval;
	struct task_struct *p;

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

	if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))
		return ERR_PTR(-EINVAL);

```

函数**copy_process**()内核源码如下：

```c
static __latent_entropy struct task_struct *copy_process(
					unsigned long clone_flags,
					unsigned long stack_start,
```

```
                unsigned long stack_size,
                int __user *child_tidptr,
                struct pid *pid,
                int trace,
                unsigned long tls,
                int node)
{
    int retval;
    struct task_struct *p;
    if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEW
NS|CLONE_FS))
        return ERR_PTR(-EINVAL);
    if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NE
WUSER|CLONE_FS))
        return ERR_PTR(-EINVAL);
    /*
     * Thread groups must share signals as well, and detached thread
s
     * can only be started up within the thread group.
     */
    if ((clone_flags & CLONE_THREAD) && !
(clone_flags & CLONE_SIGHAND))
        return ERR_PTR(-EINVAL);
    /*
     * Shared signal handlers imply shared VM. By way of the above,
     * thread groups also imply shared VM. Blocking this case allows
     * for various simplifications in other code.
```

```
	 */
	if ((clone_flags & CLONE_SIGHAND) && !
(clone_flags & CLONE_VM))
		return ERR_PTR(-EINVAL);
	/*
	 * Siblings of global init remain as zombies on exit since they are
	 * not reaped by their parent (swapper). To solve this and to avoi
d
	 * multi-rooted process trees, prevent global and container-inits
	 * from creating siblings.
	 */
	if ((clone_flags & CLONE_PARENT) &&
				current->signal->flags & SIGNAL_UNKILLABLE)
		return ERR_PTR(-EINVAL);
	/*
	 * If the new process will be in a different pid or user namespace
	 * do not allow it to share a thread group with the forking task.
	 */
	if (clone_flags & CLONE_THREAD) {
		if ((clone_flags & (CLONE_NEWUSER | CLONE_NEWPID)) ||
			(task_active_pid_ns(current) !=
				current->nsproxy->pid_ns_for_children))
			return ERR_PTR(-EINVAL);
	}
	retval = security_task_create(clone_flags);
	if (retval)
		goto fork_out;
```

```
	retval = -ENOMEM;
	p = dup_task_struct(current, node);
	if (!p)
		goto fork_out;
	/*
	 * This _must_ happen before we call free_task(), i.e. before we ju
mp
	 * to any of the bad_fork_* labels. This is to avoid freeing
	 * p-
>set_child_tid which is (ab)used as a kthread's data pointer for
	 * kernel threads (PF_KTHREAD).
	 */
	p-
>set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr
 : NULL;
	/*
	 * Clear TID on mm_release()?
	 */
	p-
>clear_child_tid = (clone_flags & CLONE_CHILD_CLEARTID) ? child_t
idptr : NULL;
	ftrace_graph_init_task(p);
	rt_mutex_init_task(p);
#ifdef CONFIG_PROVE_LOCKING
	DEBUG_LOCKS_WARN_ON(!p->hardirqs_enabled);
	DEBUG_LOCKS_WARN_ON(!p->softirqs_enabled);
#endif
```

```
	retval = -EAGAIN;
	if (atomic_read(&p->real_cred->user->processes) >=
			task_rlimit(p, RLIMIT_NPROC)) {
		if (p->real_cred->user != INIT_USER &&
		    !capable(CAP_SYS_RESOURCE) && !capable(CAP_SYS_ADMI
N))
			goto bad_fork_free;
	}
	current->flags &= ~PF_NPROC_EXCEEDED;
	retval = copy_creds(p, clone_flags);
	if (retval < 0)
		goto bad_fork_free;
	/*
	 * If multiple threads are within copy_process(), then this check
	 * triggers too late. This doesn't hurt, the check is only there
	 * to stop root fork bombs.
	 */
	retval = -EAGAIN;
	if (nr_threads >= max_threads)
		goto bad_fork_cleanup_count;
	delayacct_tsk_init(p);	/* Must remain after dup_task_struct() */
	p->flags &= ~(PF_SUPERPRIV | PF_WQ_WORKER | PF_IDLE);
	p->flags |= PF_FORKNOEXEC;
	INIT_LIST_HEAD(&p->children);
	INIT_LIST_HEAD(&p->sibling);
	rcu_copy_process(p);
	p->vfork_done = NULL;
```

```c
	spin_lock_init(&p->alloc_lock);
	init_sigpending(&p->pending);
	p->utime = p->stime = p->gtime = 0;
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
	p->utimescaled = p->stimescaled = 0;
#endif
	prev_cputime_init(&p->prev_cputime);
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
	seqcount_init(&p->vtime_seqcount);
	p->vtime_snap = 0;
	p->vtime_snap_whence = VTIME_INACTIVE;
#endif
#if defined(SPLIT_RSS_COUNTING)
	memset(&p->rss_stat, 0, sizeof(p->rss_stat));
#endif
	p->default_timer_slack_ns = current->timer_slack_ns;
	task_io_accounting_init(&p->ioac);
	acct_clear_integrals(p);
	posix_cpu_timers_init(p);
	p->start_time = ktime_get_ns();
	p->real_start_time = ktime_get_boot_ns();
	p->io_context = NULL;
	p->audit_context = NULL;
	cgroup_fork(p);
#ifdef CONFIG_NUMA
	p->mempolicy = mpol_dup(p->mempolicy);
	if (IS_ERR(p->mempolicy)) {
```

```
        retval = PTR_ERR(p->mempolicy);
        p->mempolicy = NULL;
        goto bad_fork_cleanup_threadgroup_lock;
    }
#endif
#ifdef CONFIG_CPUSETS
    p->cpuset_mem_spread_rotor = NUMA_NO_NODE;
    p->cpuset_slab_spread_rotor = NUMA_NO_NODE;
    seqcount_init(&p->mems_allowed_seq);
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
    p->irq_events = 0;
    p->hardirqs_enabled = 0;
    p->hardirq_enable_ip = 0;
    p->hardirq_enable_event = 0;
    p->hardirq_disable_ip = _THIS_IP_;
    p->hardirq_disable_event = 0;
    p->softirqs_enabled = 1;
    p->softirq_enable_ip = _THIS_IP_;
    p->softirq_enable_event = 0;
    p->softirq_disable_ip = 0;
    p->softirq_disable_event = 0;
    p->hardirq_context = 0;
    p->softirq_context = 0;
#endif
    p->pagefault_disabled = 0;
#ifdef CONFIG_LOCKDEP
```

```c
    p->lockdep_depth = 0; /* no locks held yet */
    p->curr_chain_key = 0;
    p->lockdep_recursion = 0;
#endif
#ifdef CONFIG_DEBUG_MUTEXES
    p->blocked_on = NULL; /* not blocked yet */
#endif
#ifdef CONFIG_BCACHE
    p->sequential_io    = 0;
    p->sequential_io_avg    = 0;
#endif
    /* Perform scheduler related setup. Assign this task to a CPU. */
    retval = sched_fork(clone_flags, p);
    if (retval)
        goto bad_fork_cleanup_policy;
    retval = perf_event_init_task(p);
    if (retval)
        goto bad_fork_cleanup_policy;
    retval = audit_alloc(p);
    if (retval)
        goto bad_fork_cleanup_perf;
    /* copy all the process information */
    shm_init_task(p);
    retval = security_task_alloc(p, clone_flags);
    if (retval)
        goto bad_fork_cleanup_audit;
    retval = copy_semundo(clone_flags, p);
```

```c
if (retval)
    goto bad_fork_cleanup_security;
retval = copy_files(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_semundo;
retval = copy_fs(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_files;
retval = copy_sighand(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_fs;
retval = copy_signal(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_sighand;
retval = copy_mm(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_signal;
retval = copy_namespaces(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_mm;
retval = copy_io(clone_flags, p);
if (retval)
    goto bad_fork_cleanup_namespaces;
retval = copy_thread_tls(clone_flags, stack_start, stack_size, p, tls);
if (retval)
    goto bad_fork_cleanup_io;
if (pid != &init_struct_pid) {
```

```
		pid = alloc_pid(p->nsproxy->pid_ns_for_children);
		if (IS_ERR(pid)) {
			retval = PTR_ERR(pid);
			goto bad_fork_cleanup_thread;
		}
	}
#ifdef CONFIG_BLOCK
	p->plug = NULL;
#endif
#ifdef CONFIG_FUTEX
	p->robust_list = NULL;
#ifdef CONFIG_COMPAT
	p->compat_robust_list = NULL;
#endif
	INIT_LIST_HEAD(&p->pi_state_list);
	p->pi_state_cache = NULL;
#endif
	/*
	 * sigaltstack should be cleared when sharing the same VM
	 */
	if ((clone_flags & (CLONE_VM|CLONE_VFORK)) == CLONE_VM)
		sas_ss_reset(p);
	/*
	 * Syscall tracing and stepping should be turned off in the
	 * child regardless of CLONE_PTRACE.
	 */
	user_disable_single_step(p);
```

```
	clear_tsk_thread_flag(p, TIF_SYSCALL_TRACE);
#ifdef TIF_SYSCALL_EMU
	clear_tsk_thread_flag(p, TIF_SYSCALL_EMU);
#endif
	clear_all_latency_tracing(p);
	/* ok, now we should be set up.. */
	p->pid = pid_nr(pid);
	if (clone_flags & CLONE_THREAD) {
		p->exit_signal = -1;
		p->group_leader = current->group_leader;
		p->tgid = current->tgid;
	} else {
		if (clone_flags & CLONE_PARENT)
			p->exit_signal = current->group_leader->exit_signal;
		else
			p->exit_signal = (clone_flags & CSIGNAL);
		p->group_leader = p;
		p->tgid = p->pid;
	}
	p->nr_dirtied = 0;
	p->nr_dirtied_pause = 128 >> (PAGE_SHIFT - 10);
	p->dirty_paused_when = 0;
	p->pdeath_signal = 0;
	INIT_LIST_HEAD(&p->thread_group);
	p->task_works = NULL;
	cgroup_threadgroup_change_begin(current);
	/*
```

```
	 * Ensure that the cgroup subsystem policies allow the new process to be
	 * forked. It should be noted the the new process's css_set can be changed
	 * between here and cgroup_post_fork() if an organisation operation is in
	 * progress.
	 */
	retval = cgroup_can_fork(p);
	if (retval)
		goto bad_fork_free_pid;
	/*
	 * Make it visible to the rest of the system, but dont wake it up yet.
	 * Need tasklist lock for parent etc handling!
	 */
	write_lock_irq(&tasklist_lock);
	/* CLONE_PARENT re-uses the old parent */
	if (clone_flags & (CLONE_PARENT|CLONE_THREAD)) {
		p->real_parent = current->real_parent;
		p->parent_exec_id = current->parent_exec_id;
	} else {
		p->real_parent = current;
		p->parent_exec_id = current->self_exec_id;
	}
	klp_copy_process(p);
	spin_lock(&current->sighand->siglock);
```

```
    /*
     * Copy seccomp details explicitly here, in case they were change
d
     * before holding sighand lock.
     */
    copy_seccomp(p);
    /*
     * Process group and session signals need to be delivered to just
the
     * parent before the fork or both the parent and the child after th
e
     * fork. Restart if a signal comes in before we add the new proces
s to
     * it's process group.
     * A fatal signal pending means that current will exit, so the new
     * thread can't slip out of an OOM kill (or normal SIGKILL).
    */
    recalc_sigpending();
    if (signal_pending(current)) {
        retval = -ERESTARTNOINTR;
        goto bad_fork_cancel_cgroup;
    }
    if (unlikely(!(ns_of_pid(pid)-
>nr_hashed & PIDNS_HASH_ADDING))) {
        retval = -ENOMEM;
        goto bad_fork_cancel_cgroup;
    }
```

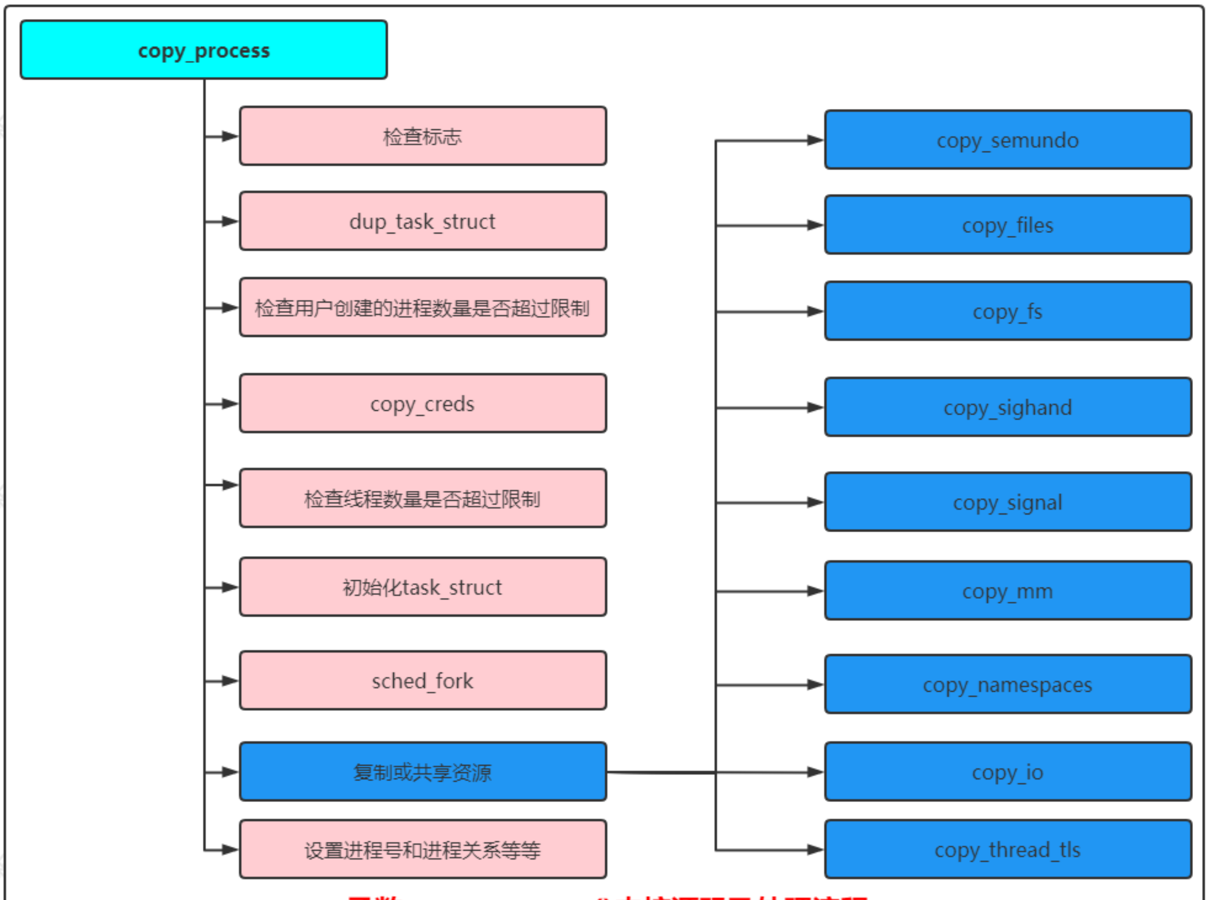
```
    if (likely(p->pid)) {
        ptrace_init_task(p, (clone_flags & CLONE_PTRACE) || trace);
        init_task_pid(p, PIDTYPE_PID, pid);
        if (thread_group_leader(p)) {
            init_task_pid(p, PIDTYPE_PGID, task_pgrp(current));
            init_task_pid(p, PIDTYPE_SID, task_session(current));
            if (is_child_reaper(pid)) {
                ns_of_pid(pid)->child_reaper = p;
                p->signal->flags |= SIGNAL_UNKILLABLE;
            }
            p->signal->leader_pid = pid;
            p->signal->tty = tty_kref_get(current->signal->tty);
            /*
             * Inherit has_child_subreaper flag under the same
             * tasklist_lock with adding child to the process tree
             * for propagate_has_child_subreaper optimization.
             */
            p->signal->has_child_subreaper = p->real_parent->signal-
>has_child_subreaper ||
                            p->real_parent->signal->is_child_subreaper;
            list_add_tail(&p->sibling, &p->real_parent->children);
            list_add_tail_rcu(&p->tasks, &init_task.tasks);
            attach_pid(p, PIDTYPE_PGID);
            attach_pid(p, PIDTYPE_SID);
            __this_cpu_inc(process_counts);
        } else {
            current->signal->nr_threads++;
```

```
			atomic_inc(&current->signal->live);
			atomic_inc(&current->signal->sigcnt);
			list_add_tail_rcu(&p->thread_group,
					  &p->group_leader->thread_group);
			list_add_tail_rcu(&p->thread_node,
					  &p->signal->thread_head);
		}
		attach_pid(p, PIDTYPE_PID);
		nr_threads++;
	}
	total_forks++;
	spin_unlock(&current->sighand->siglock);
	syscall_tracepoint_update(p);
	write_unlock_irq(&tasklist_lock);
	proc_fork_connector(p);
	cgroup_post_fork(p);
	cgroup_threadgroup_change_end(current);
	perf_event_fork(p);
	trace_task_newtask(p, clone_flags);
	uprobe_copy_process(p, clone_flags);
	return p;
bad_fork_cancel_cgroup:
	spin_unlock(&current->sighand->siglock);
	write_unlock_irq(&tasklist_lock);
	cgroup_cancel_fork(p);
bad_fork_free_pid:
	cgroup_threadgroup_change_end(current);
```

```
    if (pid != &init_struct_pid)
        free_pid(pid);
bad_fork_cleanup_thread:
    exit_thread(p);
bad_fork_cleanup_io:
    if (p->io_context)
        exit_io_context(p);
bad_fork_cleanup_namespaces:
    exit_task_namespaces(p);
bad_fork_cleanup_mm:
    if (p->mm)
        mmput(p->mm);
bad_fork_cleanup_signal:
    if (!(clone_flags & CLONE_THREAD))
        free_signal_struct(p->signal);
bad_fork_cleanup_sighand:
    __cleanup_sighand(p->sighand);
bad_fork_cleanup_fs:
    exit_fs(p); /* blocking */
bad_fork_cleanup_files:
    exit_files(p); /* blocking */
bad_fork_cleanup_semundo:
    exit_sem(p);
bad_fork_cleanup_security:
    security_task_free(p);
bad_fork_cleanup_audit:
    audit_free(p);
```

```
bad_fork_cleanup_perf:
    perf_event_free_task(p);
bad_fork_cleanup_policy:

// 配置NUMA(NUMA(Non Uniform Memory Access)即非一致内存访问架构，市面上主要有X86_64(JASPER)和MIPS64(XLP)体系。)
#ifdef CONFIG_NUMA
    mpol_put(p->mempolicy);
bad_fork_cleanup_threadgroup_lock:
#endif
    delayacct_tsk_free(p);
bad_fork_cleanup_count:
    atomic_dec(&p->cred->user->processes);
    exit_creds(p);
bad_fork_free:
    p->state = TASK_DEAD;
    put_task_stack(p);
    free_task(p);
fork_out:
    return ERR_PTR(retval);
}
```

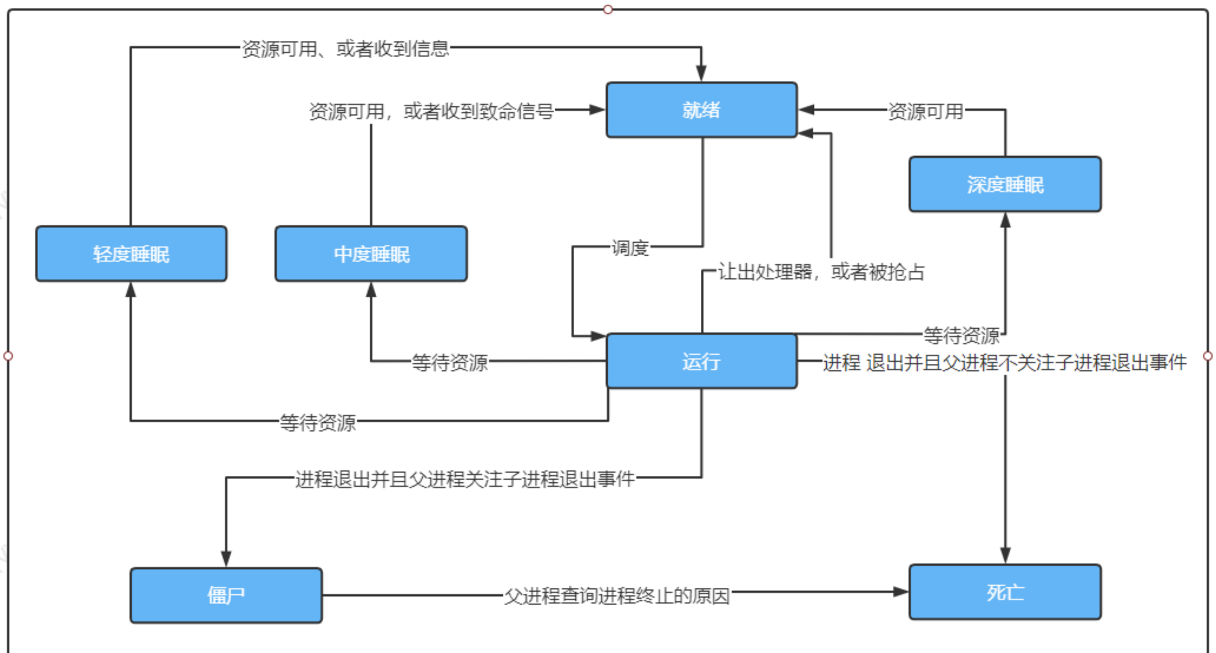
函数copy_process()：创建新进程的主要工作由此函数完成，具体处理流程如下图所示：



函数copy_process()内核源码及处理流程

二、进程状态迁移

进程主要有7种状态：就绪状态、运行状态、轻度睡眠、中度睡眠、深度睡眠、僵尸状态、死亡状态，它们之间状态变迁如下：



进程状态迁移

三、调度策略及优先级

1、Linux内核支持调度策略

- 先进先出调度（SCHED_FIFO）、轮流调度（SCHED_RR）、限期调度策略（SCHED_DEADLINE)采用不同的调度策略调度实时进程。
- 普通进程支持两种调度策略：标准轮流分时（SCHED_NORMAL）和SCHED_BATCH调度普通的非实时进程。
- 空闲（SCHED_IDLE）则在系统空闲时调用idle进程。

2、进程优先级

限期进程的优先级比实时进程高，实时进程的优先级比普通进程高。

- 限制进程的优先级是-1。
- 实时进程的褚优先级是1-99，优先级数值越大，表示优先级越高。
- 普通进程 的静态优先级是100-139，优先级值越小，表示优先级越高，可通过修改nice值改变普通进程 的优先级，优先级等于120加上nice值。

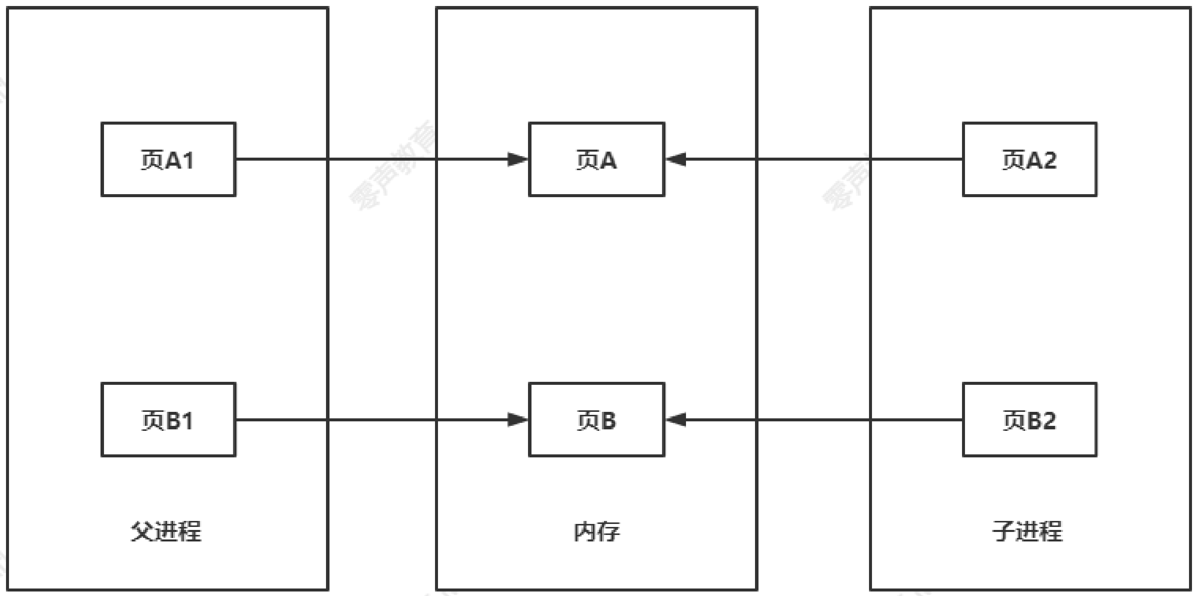
在task_struct结构体中，4个成员和优先级有关如下：

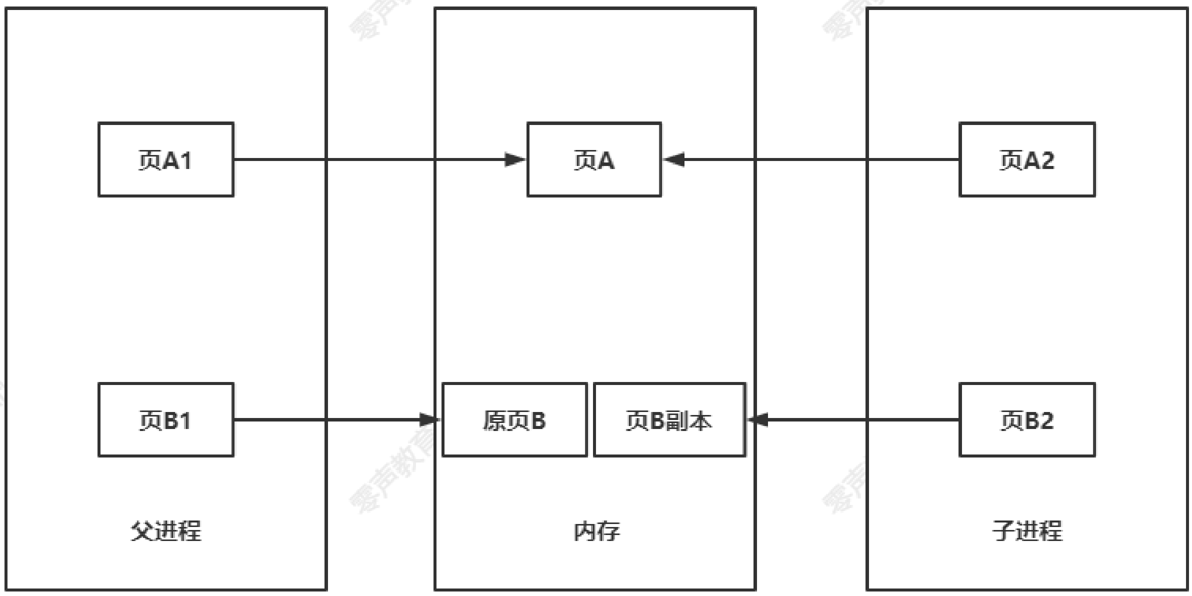
include > linux > C sched.h > task_struct

```c
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state;
	void				*stack;
	atomic_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;

#ifdef CONFIG_SMP
	struct llist_node		wake_entry;
	int				on_cpu;
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/* Current CPU: */
	unsigned int			cpu;
#endif
	unsigned int			wakee_flips;
	unsigned long			wakee_flip_decay_ts;
	struct task_struct		*last_wakee;

	int				wake_cpu;
#endif
	int				on_rq;

	int				prio;
	int				static_prio;
	int				normal_prio;
	unsigned int			rt_priority;

	const struct sched_class	*sched_class;
```

四、写时复制

写时复制核心思想：只有在不得不复制数据内容时才去复制数据内容。


页A1
页A
页A2
页B1
页B
页B2
父进程
内存
子进程


页A1
页A
页A2
页B1
原页B
页B副本
页B2
父进程
内存
子进程